MINIMIZING THE IMPACT OF SYNCHRONIZATION OVERHEAD IN PARALLEL DISCRETE EVENT SIMULATIONS

THESIS

Andrew Christopher Walton
Second Lieutenant, USAF

AFIT/GCS/ENG/94D-25

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/94D-25

MINIMIZING THE IMPACT OF SYNCHRONIZATION OVERHEAD IN PARALLEL DISCRETE EVENT SIMULATIONS

THESIS

Andrew Christopher Walton
Second Lieutenant, USAF

AFIT/GCS/ENG/94D-25

Approved for public release; distribution unlimited

AFIT/GCS/ENG/94D-25

MINIMIZING THE IMPACT OF SYNCHRONIZATION OVERHEAD IN PARALLEL DISCRETE EVENT SIMULATIONS

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Andrew Christopher Walton, B.S.E.E.

Second Lieutenant, United States Air Force

14 December, 1994

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1991	Master's Thesis

4. TITLE AND SUBTITLE

Minimizing the Impact of Synchronization Overhead in Parallel Discrete Event Simulations

5. FUNDING NUMBERS

6. AUTHOR(S)

Andrew C. Walton

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/93D-25

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

ARPA/CSTO, 4001 N. Fairfax Dr. #200, Arlington, VA 22203-1615

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A Parallel Discrete Event Simulation Coprocessor was designed for conservative synchronization protocols and was implemented in software using some of a parallel computer's nodes to act as coprocessors. The coprocessor was designed to offload synchronization overhead and next event queue management from the nodes running the simulation. The coprocessor was designed to accelerate simulations based on the Simulation Protocol Evaluation on a Concurrent Testbed with ReUsable Modules (SPECTRUM) environment. The research was conducted in three steps: the SPECTRUM environment was ported from an Intel iPSC/2 to an Intel Paragon XP/S, the coprocessor was designed and the simulations were timed, with and without the coprocessor. In some cases, the coprocessor provided up to a 2.5 times speedup. On other simulations, the coprocessor slowed the simulation a small amount. This reduction in speed was due to communication delays between the logical processes and the coprocessors that were incurred by placing them on separate nodes. The communications delay was accurately modeled for a simple simulation and spinloops were used to compensate for the delay. The delay would be several orders of magnitude smaller if the coprocessor was actually implemented in hardware. The simulations that were not accelerated by the coprocessor were not being slowed by null message passing or next event queue management. Instead, these simulations were slowed by blocking times where the logical processes were forced to wait for a message from another logical process that would allow them to continue the simulation. This research concluded that parallel simulations need to be partitioned to logical processes in a manner which reduces blocking times.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Parallel Simulation, Coprocessor, Conservative Synchronization	143
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Acknowledgments

Several faculty members at AFIT made significant contributions to my research. Lt Col Tom S. Wailes guided me in the right direction by helping me decide that the coprocessor could be implemented more quickly in software than hardware. I am very thankful for his guidance and encouragement. Lt Col William C. Hobart Jr., my thesis advisor, provided innumerable insights into how parallel discrete event simulations work and helped me learn to program using an unfamiliar language and operating system. Lt Col Hobart also helped me focus my attention on only the important issues; he kept me from becoming side-tracked on several occasions.

My wife's contributions to my thesis were also critical to its success. Christina proofread my thesis several times and helped me make hundreds of corrections. Her understanding and patience helped make our first year and a half of marriage a pleasant one, even with the demands of classwork and research. I am excited about our future together.

Most of all, I would like to thank my Lord and Savior, Jesus Christ for answering my prayers and giving me the strength I needed to complete the tasks set before me. My Redeemer is faithful and true.

Andrew Christopher Walton

Table of Contents

Page

List of Figures

List of Tables

Abstract

A Parallel Discrete Event Simulation Coprocessor was designed for conservative synchronization protocols and was implemented in software using some of a parallel computer's nodes to act as coprocessors. The coprocessor was designed to offload synchronization overhead and next event queue management from the nodes running the simulation. The coprocessor was designed to accelerate simulations based on the Simulation Protocol Evaluation on a Concurrent Testbed with ReUsable Modules (SPECTRUM) environment. The research was conducted in three steps: the SPECTRUM environment was ported from an Intel iPSC/2 to an Intel Paragon XP/S, the coprocessor was designed and the simulations were timed, with and without the coprocessor. In some cases, the coprocessor provided up to a 2.5 times speedup. On other simulations, the coprocessor slowed the simulation a small amount. This reduction in speed was due to communication delays between the logical processes and the coprocessors that were incurred by placing them on separate nodes. The communications delay was accurately modeled for a simple simulation and spinloops were used to compensate for the delay. The delay would be several orders of magnitude smaller if the coprocessor was actually implemented in hardware. The simulations that were not accelerated by the coprocessor were not being slowed by null message passing or next event queue management. Instead, these simulations were slowed by blocking times where the logical processes were forced to wait for a message from another logical process that would allow them to continue the simulation. This research concluded that parallel simulations need to be partitioned to logical processes in a manner which reduces blocking times.

MINIMIZING THE IMPACT OF SYNCHRONIZATION OVERHEAD IN PARALLEL DISCRETE EVENT SIMULATIONS

I. Background and Statement of Problem

1.1 Background

Computer simulations are useful for modeling a wide range of physical systems. Many of the objects and systems that the Department of Defense needs to simulate, such as battlefields and electronic circuits, are very complex and can consume large amounts of computer simulation time. The purpose of this research is to characterize the parallelism in a specific type of simulation, called Parallel Discrete Event Simulations (PDES), and use this characterization to determine how a coprocessor could accelerate such simulations.

The goal of parallel discrete event simulations is to accelerate simulations by running parts of the simulation in parallel. Running the simulations in parallel allows many processors to work on the same task simultaneously and speed up the task's completion. Amdahl's law states that the performance improvement gained from using a faster mode of execution is limited by the portion of the time that the faster mode can be used [1:8]. The speedup provided by PDES can be calculated using the following formula [1:9]:

$$\text{Speedup} = \frac{\text{Execution time of task without using the enhancement}}{\text{Execution time of the task using the enhancement when possible}}$$

Equation 1: Speedup

For PDES, the execution time of the task without the enhancement would be the time that it took for the fastest known sequential algorithm to run. The execution time of the task using the enhancement would be the time that it took the parallel simulation to run. This formula implies that the acceleration of the simulation is proportional to how much of the simulation can be run in parallel. The more that a simulation is able to exploit parallelism, the faster it will run. Parallel simulations, by nature, contain significant amounts of parallelism, but this parallelism is extremely difficult to exploit [2].

1.2 Problem Statement

Parallel discrete event simulations rely on the assumption that events only occur at discrete points in time. Like sequential discrete event simulations, PDES schedule events in next event queues and process those events in the time-order they are scheduled. The simulations take the earliest scheduled event off of the top of the queue, process it and schedule any new events. Unlike sequential simulations, PDES have multiple next event queues, one for each process running the simulation. To divide the simulations' events so that they can be run in parallel, the physical system being modeled is broken down into physical processes which interact with each other through messages[3:198]. Each physical process corresponds to a Logical Process (LP) which is run on a node of the parallel computer. In addition to processing the events in its local next event queue, each LP must also schedule events, by passing messages to be processed by other LPs in the system and remain synchronized with other LPs running the simulation. An event

scheduled for the future which affects an event scheduled in the past causes what is called a causality error. The synchronization mechanism used in the simulation must prevent or correct causality errors without creating an excessive load on the LP's processor. If the synchronization mechanism takes too much processor time, the LP will run more slowly and the speedup will be adversely affected. An inefficient synchronization mechanism can cause the speedup to be less than one, so that the simulation takes longer on a parallel machine than on a sequential machine.

Apart from the parallelism inherent in running the simulation on multiple nodes, there is some parallelism in the tasks performed by the nodes themselves. In addition to performing synchronization tasks, each LP has to manage its local next event queue. These tasks increase the load on the processor and reduce the amount of parallelism that can be exploited. This research examines how much parallelism can be exploited by placing all of the synchronization tasks on a coprocessor and examines the coprocessor's effect on the LP's workload.

1.3 Summary of Current Knowledge

This research has been preceded by several other theses at AFIT. Taylor proposed a design for a Discrete Event Simulation Coprocessor which managed synchronization and message-passing tasks for the host processor. This coprocessor would provide a speedup between 1 and 20 as the number of LP's per node varied. Daniel continued work on the coprocessor by modeling it at the gate-level using the VHSIC Hardware Description Language (VHDL). Using VHDL simulations, Daniel's design tested the coprocessor's feasibility and predicted that the coprocessor could provide a speedup between 1.3 and 60, depending on the granularity of the simulation[4:86]. Berlin refined the coprocessor's design, breaking it up into components that were implemented using Field Programmable Gate Arrays (FPGA), custom-fabricated circuits,

and commercial products. Berlin fabricated and tested many of the custom circuits, using the observed behavior of several simulations to design realistic tests. Using a VHDL simulation of a Wallace Tree multiplier, Berlin obtained a speedup of four times for the next event queues' functions that the coprocessor performed. Unfortunately, he also found that the acceleration of the next event queue would result in only a 1.02 speedup since the queue management functions took less than 2% of the application's time[5:68].

Berlin's results deviate greatly from the expected results because of the manner in which the parts of the simulations were timed. The time that it took for an LP to get an event from the next event queue was greatly overstated. It included the length of time that the LP was blocked, waiting for either an event or a synchronization message from another LP. Since many of the simulations tend to be communication bound, the LPs spent a significant amount of time waiting for messages. The process of getting an event was originally considered the most time-consuming, but is actually very small. Since the next event queues in the parallel simulations at AFIT are implemented with linked lists, the processor requesting an event simply looks at the head of the linked list to get the next event. Getting an event is, therefore, much quicker than placing an event in the queue.

Much of the parallelism derived from the use of the coprocessor will result from the coprocessor's ability to place events in the next event queue and perform synchronization tasks while the LP continues to process events. To insert an event in the next event queue, the processor will have to traverse the linked list until it finds the correct time slot for the event it is trying to insert. In the worst case, the coprocessor will have to traverse the entire linked list to find the correct position for the message. Since the messages from other LPs tend to be in the future relative to the receiving LP's clock, it makes sense to insert them into the list, traversing the list

from the tail, and minimizing the likelihood that the coprocessor will have to traverse the entire list.

Since Berlin's results indicated that the coprocessor would produce an insignificant amount of speedup, it is necessary to compare the capability of the coprocessor with a microprocessor performing the same task with software. If a general purpose microprocessor could provide more useful functions to the LP, it would be feasible to use some of the nodes of the parallel computer as coprocessors. In the hardware for the coprocessor, the next event queue (NEQ) is implemented using an Extreme Search Associative Memory (ESAM) which will allow insertions and removals to occur with O(1) complexity. The O(1) complexity of the NEQ operations is a result of the properties of the ESAM, which uses hardware to search for the event with the lowest time-stamp. For a general purpose microprocessor, using a linked list to implement the queue, the order-of complexity for NEQ operations is also O(1) for getting an event from the queue and O(n) for placing events on the queue, where n is the number of events stored in the queue. Although the general purpose microprocessor appears to be slower than the coprocessor at inserting events into the list, it is important to remember that any order-of analysis neglects a constant factor by which the order-of analysis is multiplied. This means that for a small n, O(n) operations could actually be faster than O(1) operations if the constant in front of the O(n) is small and the constant in front of the O(1) is large. It is difficult to measure the constants in this order-of analysis, but we can estimate their size by looking at the properties of the ESAM and of a general-purpose microprocessor.

Berlin's timing analysis provided enough information to calculate the approximate times it would take to perform the various operations of the coprocessor. By calculating these times, this research determined how fast the coprocessor was compared to a microprocessor. Berlin's

equations for the coprocessor's timing required the following assumptions about the simulation's characteristics:

- The number of input arcs would be seven.
- The number of output arcs would be seven.
- 90% of the messages would be written to reserved words in the ESAM
- 10% of the messages would be written to unreserved words in the ESAM
- 80% of the messages would be null messages.
- 20% of all of the messages would be real messages.
- 10% of the time, the LP status would need to be updated after Get Event

The above assumptions were intentionally chosen in an optimistic manner and the coprocessor would probably not perform quite as well as the equations predict that it will perform. The number of input arcs and the number of output arcs were chosen to be the same number of arcs the processor would have if connected to the Parallel Simulation Group's Intel iPSC/2 Hypercube.

The clock frequency of the coprocessor chip was determined by Berlin's analysis of its critical path. The ESAM was in the coprocessor's critical path and limited the clock frequency to 8.7 MHz. Since the ESAM's critical path increased linearly as additional words were added to it, the coprocessor's clock frequency would be decreased to an even slower rate when the coprocessor's memory was expanded.

The times for the coprocessor's operations are shown in Table 1 below:

Routine	Clock Cycles Required	Time Required (microseconds) (assumes 8.7 MHz clock)
Initialize Simulation	1844	211.954
Post Message	420.4	48.3218
Get Event	974.8	112.046
Post Event	792	91.0345

Table 1: Times Required for Coprocessor Operations

The times for each operation would take were fairly long, as Table 1 suggests, and would limit the practicality of the original design of the coprocessor. The above timing results indicate that a general purpose microprocessor could perform the functions of the Discrete Event Simulation Coprocessor more efficiently. In addition to increased speedup, the benefits of using a general purpose microprocessor appear to greatly out-weigh the value of a specialized chip for message-synchronization. Some of these benefits are:

- A general purpose microprocessor would be less risky since it is an off-the-shelf part.
- It would be less expensive than a custom-fabricated chip.
- It would be easier to increase the size of the queue in the microprocessor than it would be to scale up the amount of memory in the ESAM.
- The microprocessor could store all of the message's data where the Discrete Event Simulation Coprocessor could only store a pointer to the message in the host processor's memory.

- A microprocessor would be more flexible because its software can be changed without removing and replacing chips. (A wide variety of simulations could be more easily accommodated.)

- A microprocessor would be faster than the Discrete Event Simulation Coprocessor.

These benefits justified this research's attempt to use a general purpose microprocessor to emulate the coprocessor. The decision to use a microprocessor greatly changed the nature of the research. Instead of designing and interfacing a coprocessor, a microprocessor would be programmed act as the coprocessor. To avoid having to interface another microprocessor to each of the parallel computer's nodes, half of the nodes would be used to run the actual simulation, and the other half of the nodes would be configured to act as the coprocessors. This configuration of the computer would allow the coprocessor to be simulated using software.

To use half of a parallel computer's nodes as coprocessors, software had to be written that made the nodes behave like the coprocessor, and modifications had to be made to Simulation Protocol Evaluation on a Concurrent Testbed with ReUsable Modules (SPECTRUM) [6], the testbed for the parallel simulation protocols. By obtaining the amount of speedup gained by different coprocessor configurations, this research could more accurately specify what type of functions needed to be implemented in a parallel discrete event simulation coprocessor.

1.4 Assumptions

Several assumptions significantly affected the architecture of this design. One of the biggest assumptions concerned the architecture of the Intel Paragon: the simulation would be computation-bound and the time it would take to pass messages between the Paragon's nodes

would be very small compared to the time it took the software coprocessors to complete their tasks. If the Paragon took a long time to pass messages between the nodes running the simulation and the nodes running the coprocessor software, then the messages passed between those nodes would be a bottleneck. If the simulations were too fine grained, and were communications-bound, the simulation would run faster without the coprocessor and the speedup would be less than one. Since the carwash simulation does very little computation, it was suspected that the carwash would not be accelerated much, if any, by the coprocessor. The VSIM simulations, however, appear to be coarser grained and it appeared that they might be accelerated if the synchronization and SPECTRUM next event queue management were off loaded to the coprocessor. This research also assumed that the simulation's granularity could be increased by adding spin loops to the parts of the simulations that process events. By varying the duration of the spin loops, this research could characterize the granularity at which the coprocessor provided the most speedup.

This research also assumed that SPECTRUM could be modified without having to make significant changes to the Carwash Simulation and VSIM. If a lot of time was spent modifying the simulations, it would be difficult to test different configurations of the coprocessor. For this reason, the coprocessor is designed to appear transparent to the application running on SPECTRUM.

1.5 Scope

The scope of this research was limited to implementing several different coprocessor designs in software and characterizing their performance. The coprocessor runs on separate nodes of the parallel computer, with one coprocessor per LP. The standard carwash simulation and VHDL simulations were used to characterize the performance of the coprocessors. The VHDL

simulations were run using VSIM and simulate the same VHDL files that Berlin used to characterize his design.

1.6 Approach

This research was completed in five steps: the simulations were ported to the Intel Paragon, the coprocessor was designed, the coprocessor was implemented, its effect on the simulation's performance was measured, and its performance was analyzed. The simulations were originally designed to run on the Intel iPSC/2 Hypercube in AFIT's Parallel Simulation Lab. They used SPECTRUM to perform the synchronization tasks necessary to prevent causality errors. SPECTRUM allows different synchronization mechanisms to be tested without having to alter the application. The first step in porting the simulations was to port SPECTRUM to the Paragon since it contained all of the machine-specific code. Then, by modifying the simulations' make files, they would run on the Paragon.

Once SPECTRUM ran on the Paragon, this research examined ways it could exploit parallelism in the activities an individual node performed. It determined what functions could be placed on the coprocessor and then implemented them. After finishing the coprocessor's implementation, its performance was compared to the simulations' performance without the coprocessor. The analysis of the results of this comparison revealed many of the effects of synchronization communication latency and the effects of the tasks' granularity.

1.7 Outline of Thesis

The rest of this document is organized as follows: The second chapter is a review of related theses and technical literature. The third chapter describes how SPECTRUM was ported to the Intel Paragon XP/S, and the fourth chapter discusses the coprocessor's design and implementation. The methods used to analyze the simulations' performance and performance measurements are described in chapter five. The sixth, and final, chapter contains conclusions and recommendations for future research.

II. Literature Review

2.1 Introduction

This chapter describes the state of current research into parallel discrete event simulations with an emphasis on accelerating them. Articles describing both PDES algorithms and hardware acceleration were examined to see if they were relevant to the design of the coprocessor. These articles provided the background necessary to complete this research.

2.2 PDES Algorithms

Parallel discrete event simulations have been implemented using a variety of algorithms. These algorithms are generally classified as being either conservative or optimistic. According to Fujimoto, the conservative approach uses some strategy to determine when it is safe for an event to be processed and, in doing so, strictly avoids causality errors. The optimistic approach allows causality errors to occur and uses a mechanism which detects the error and recovers[2]. No matter which simulation protocol is used, the protocol must prevent causality errors from affecting the simulation's results and prevent the LPs from deadlocking.

2.2.1 Conservative Algorithms

The first parallel discrete event simulations were based on conservative approaches where the protocol strictly avoided causality errors[7:33]. Before an LP can process an event in these simulations, it must first ensure that none of the LPs on its input arcs can send an earlier event.

Fujimoto emphasized the concept of lookahead in his description of conservative methods[7]. Lookahead is a logical process' ability to look ahead of the current value of the simulation clock and determine how far into the future the clock can progress before that logical process will have an event affecting other logical processes. If it sends messages to other logical processes, they can execute safely up to that time before they have to wait for messages from that logical process. Fujimoto tied the concept of lookahead to speedup when he said "Effectively exploiting the lookahead properties of the simulation appears to be the key to achieving good performance with these methods" [7].

One of the first and most prominent conservative protocols was developed by Chandy and Misra and is described in a survey called "Distributed Discrete-Event Simulation," by Jayadev Misra [8]. This article proposed that simulations be run on multi-processor computers in which the processors are connected so that they can pass messages. In the Chandy-Misra protocol, each LP sends messages along output arcs connected to other LPs and receives messages along input arcs. These messages are assumed to be delivered in the same order they were sent and must be sent in time-order along the arc. This means that once an LP has sent a message along an arc with a certain time stamp, it cannot send a message across the arc with an earlier time stamp. The time of the earliest possible event is called the safetime. The safetime is determined by taking the minimum time on the LP's input arcs after adding delays to the times of the last messages received along the arcs. These delays are called arc delays and vary for each of the input arcs. The arc delays define how much lookahead the simulation can exploit. The best performance in a parallel discrete event simulation can be obtained by partitioning the simulation onto LPs so that the arc delays are maximized and the LPs' interdependencies are minimized. The time of the last message sent along an arc is called the channel time. When the safetime is greater than the earliest time on the LP's next event queue, it is safe for the LP to execute that event. By using time-

stamped messages and safetimes, the Chandy-Misra protocol can be used to synchronize a simulation.

The Chandy-Misra protocol is designed to prevent deadlocks. Deadlock occurs when an LP is waiting for a message that will never be sent. There are many situations that can cause deadlock, such as when there are circular dependencies between nodes. To prevent deadlocks, the protocol sends null messages. According to Chandy and Misra, a null message " . . . is used to announce the absence of messages," and consists of an empty, time-stamped message[8:57]. Null messages prevent deadlock because they allow the LP that receives them to update its channel times. As the channel times are incremented, the LP's safetime will eventually be incremented and the simulation will avoid deadlock. When an LP is waiting for an event or is waiting for the safetime to be incremented, it is said to be blocked. The amount of time spent blocked contributes to the overhead incurred by running the simulation in parallel. Although they play a vital role in the Chandy-Misra protocol, null messages can exact a heavy penalty on simulation performance since they increase the number of messages handled by the computer's communication network and can cause communications bottlenecks.

2.2.2 Optimistic Algorithms

Optimistic protocols, as mentioned previously, allow causality errors to occur and are capable of detecting and recovering from them. Optimistic simulation protocols tend to be more complicated than conservative algorithms, but have several advantages over them. One advantage of optimistic protocols is that they can exploit parallelism in situations where causality errors might occur, but do not [7:40]. If a simulation can exploit such parallelism, it will run faster than a conservative algorithm since the conservative algorithm must always wait for the potential causality error to be avoided. Dynamic creation of processes and dynamic dependencies between

LPs are also easier to implement using optimistic protocols[7]. The primary difficulty with optimistic protocols is that they must remember old states to be able to recover from a causality error. These old states can consume large amounts of memory and restoring them frequently can slow the simulation down. The optimistic protocols, therefore, have a wider variance of speedups since they can run much faster or much slower than conservative protocols.

According to Fujimoto, the most prominent optimistic approach is Jefferson's time warp[7]. Time warp allows each of the logical processors to work at its own pace. If an LP gets a message with a time stamp that is lower than the current simulation time at that process, that message is called a straggler. When an LP receives a straggler, it must rollback to the time at which the straggler should have been scheduled if it had arrived on time. When a logical process backtracks, it must tell the other logical processes in the system that the messages that it sent, after the time stamp of the straggler, were bad and all of the interdependent logical processes must also lower their simulation times and send out messages saying they had to rollback. These messages telling the other LPs to backtrack are called antimessages.

Since old states can consume a large amount of memory, time warp provides a mechanism for eliminating them once they are no longer needed. Unneeded, old states are called fossil states. To determine when to eliminate fossil states, time warp occasionally computes the global virtual time (GVT) of all of the LPs. The GVT is the earliest local clock time on the simulation's LPs and a rollback cannot happen at any earlier time. All of the states earlier than the GVT are fossil states and can be eliminated. The problem with finding the GVT is that it is a global operation and is communication intensive. Righter and Walrand state that GVT can be estimated using a distributed algorithm with a time complexity of $O(n)$, where n is the number of processors running the simulation[9]. They also state that time warp requires less memory if the

GVT is computed often, but that this computation will increase processor and communication overhead. The time warp protocol has not been used in any of the simulations at AFIT, but might be in the future. To be general purpose, the discrete-event simulation coprocessor also had to be able to support time warp.

2.3 Hardware Acceleration

The hardware acceleration articles reviewed here fall into two categories: those which directly apply to PDES and those which apply to general purpose microprocessors. The articles pertaining to general purpose microprocessors are significant because any speedup in the microprocessors could result in a corresponding speedup in the simulation.

2.3.1 Hardware Acceleration of Simulations

Attempts to speedup parallel discrete event simulations have taken several approaches. One approach is to develop hardware for a particular synchronization protocol. Such a design is described in "Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp," by Fujimoto, Tsai and Gopalakrishnan [10]. This article states that time warp mechanisms are inefficient for simulations where large amounts of state information need to be stored for use in a rollback. Fujimoto showed that performance is reduced by 50% when the state information increases from a very small amount to 2000 bytes. Many significant simulations require much greater amounts of state space and would degrade the performance of time warp even further. The rollback chip can efficiently store state spaces of up to several megabytes every 100 microseconds and can perform a rollback every millisecond. These capabilities are intended to "allow parallel programs to exploit the advantages of time warp while avoiding most of the associated overheads"

[10]. The rollback chip is an interesting means of accelerating parallel discrete event simulations, but cannot be applied to protocols other than time warp.

Another approach to accelerating parallel discrete event simulations was presented by Reynolds. This approach uses hardware called a Parallel Reduction Network (PRN) that can " . . . quickly disseminate globally reduced values, such as GVT [Global Virtual Time] . . ." [11:435] The hardware for Reynold's design is shown in Figure 1 below. A globally reduced value is a

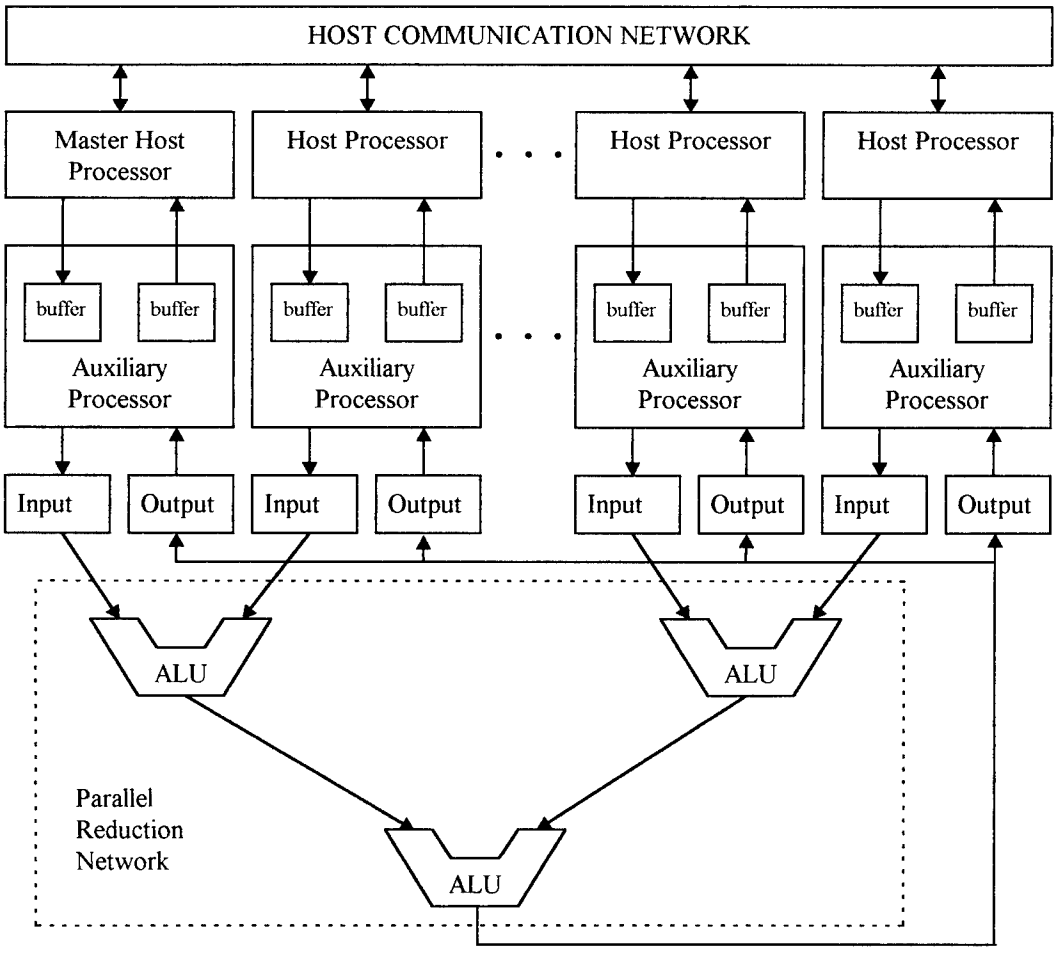

Figure 1: Parallel Reduction Network

value returned by a function that compares a variable, local to each LP, on every LP. An example of a global reduction function would be to find the minimum of some variable across all of the nodes of the simulation. The minimum value for all of the LPs' clocks is called the Global Virtual Time. Global reduction functions are also useful for such activities as lookahead computations, termination detection, and global consensus.

To be useful, a PRN would have to perform its functions much quicker than the system's existing communication network. Like the coprocessor at AFIT, Reynold's parallel reduction network can be reprogrammed and can perform a variety of tasks for different synchronization protocols. The parallel reduction network is different from AFIT's coprocessor in that it adds another means of communication to the LPs. The auxiliary processors can send data to a series of Arithmetic Logic Units (ALU) that are connected to form a binary tree. The ALUs perform a global reduction operation and return a result to the auxiliary processors. Since the ALUs are connected to form a binary tree, and the tree has a height of log n, where n is the number of processors, the PRN can perform global reduction operations in O(log n) time[11:441].

Reynolds demonstrated the PRN by using it to calculate a simulation's GVT. In such a simulation, with 8 LPs, the delay between the actual change in the GVT and the PRN's calculation of the GVT was 10-20 μs [11:450]. Since the amount of time to perform linear operations increases logarithmically as the number of LPs increases linearly, Reynolds expects operations for 32- and 64-node simulations to take the same order of magnitude of time as for 8 LPs[11:451]. Reynolds used the GVT in this simulation to perform fossil collection for a time warp protocol. A simulation using time warp can free memory used for state changes that occurred before the GVT. The PRN, however, is not limited to time warp simulations; it can be used for a variety of synchronization mechanisms. The PRN can significantly reduce the load on the computers

communications network by passing all of the information necessary to synchronize the LPs[11:451]. The main disadvantages of the parallel reduction network are that it requires an additional communication network to be added to the parallel computer and that it may not be feasible for computers with large numbers of nodes since it requires $n-1$ ALUs (where n is the number of LPs).

Comfort proposed an idea that had significant implications for the design of the hardware accelerator in "The Simulation of a Master-Slave Event Set Processor" [12]. In this research, Comfort implemented hardware which allowed NEQ management to be offloaded from a host processor to another slave processor. To facilitate this offloading of event list management, Comfort defined three primitive event list operations: schedule, next, and cancel. The schedule operation, given an event and a time, schedules that event in the NEQ. The next operation returns the event with the smallest time and increments the simulation time to that time. The cancel operation, given an event and a time, deletes the event from the NEQ, but does not alter the simulation time. By placing these operations on two or three external chips, Comfort was able to get a significant speedup. He implemented this design on a DEC PDP-11, a computer that was nearly obsolete in 1984 when he wrote the paper. To determine if his research applied to modern parallel systems, I had to determine what portion of the logical processes' time was spent managing their NEQs. The larger this time is, the more Comfort's method would accelerate the logical processes. A speedup in the execution of logical processes could result in a corresponding speedup in the overall simulation. Like Comfort, Fujimoto suggested that there was a limited amount of speedup possible from using dedicated functional units to perform some of the simulation tasks such as NEQ management[7]. This potential speedup is of great interest to this research and is investigated further in the following chapters.

2.3.2 General Purpose Accelerators

In "Processor Reconfiguration Through Instruction-Set Metamorphosis," Athanas and Silverman proposed a concept for using hardware to accelerate a microprocessor that was not directly related to simulations, but could prove useful in accelerating them[13]. This method involved adapting a general-purpose microprocessor to a specialized task by changing the way in which that microprocessor executed instructions. This is equivalent to rewriting the microprocessor's microcode automatically when software is compiled. Athanas and Silverman developed a platform called Processor Reconfiguration through Instruction Set Metamorphosis (PRISM-I). They wrote software that takes a program, written in C, and returns software and hardware image files. It then uses the hardware image files to synthesize a hardware description that can be fed into a Xilinx FPGA. The software image files are then run and use the FPGA to perform some of the computer's operations. This type of FPGA could eventually be incorporated into the design of a microprocessor, allowing the microprocessor's instructions to be customized for a particular task. One difficulty in this design is that the programmer is required to determine what instructions are being executed frequently and taking up a large percentage of the runtime. Since simulations are usually built around a simulation environment, it might be possible to optimize the simulation environment for an entire class of simulations. If a new class of simulations was developed, then the processor would only have to be reconfigured once for that whole class. Athanas and Silverman achieved speedups between 2.9 and 54 on a 10 MHz 68010-based computer system. If similar results could be achieved in the nodes of parallel computers, this method could provide a significant performance improvement to PDES.

Simulation Protocol Evaluation on a Concurrent Testbed with ReUsable Modules (SPECTRUM) was originally designed as a testbed, at the University of Virginia, to support "the empirical study of parallel simulation protocols and applications, with the expectation that experience with the testbed will provide insights into the efficacy of various protocols and their interplay with classes of applications" [14]. SPECTRUM consists of four parts: cube2.c, lp_man.c, an application, and a set of filters as shown in Figure 2 below. The file called cube2.c

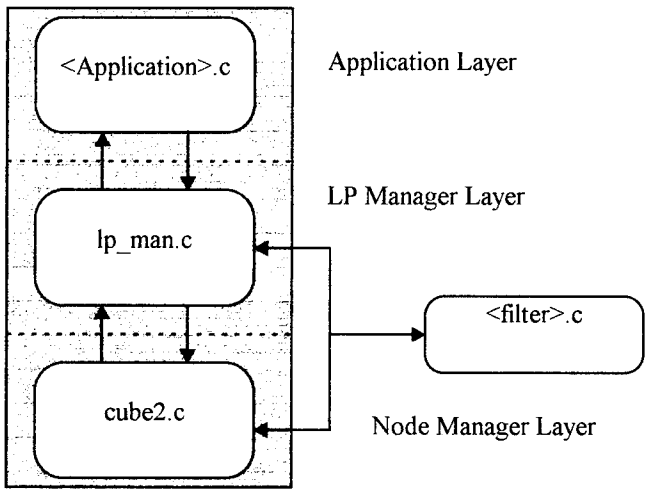

Figure 2: SPECTRUM Architecture

is SPECTRUM's interface to the parallel computer. It contains all of the system calls that are specific to the iPSC computers. To port SPECTRUM to another machine, cube2.c is the only file that needs to be changed. The file called lp_man.c is the node level manager and it performs all of the tasks of the LP. It is essentially SPECTRUM's interface to the user's application. The filters allow the functions performed by lp_man.c to be changed so that many message-passing protocols can be supported. Since the filters are the only files that need to be changed for new

protocols, the user can write a simulation and not have to rewrite it for every new protocol. To support optimistic protocols, however, the application has to be changed so that it can save its state. When a causality error occurs, the application has to restore the old state itself.

2.5 The Carwash Simulation

The carwash simulation is a simple queuing model of a carwash and has been partitioned into eight Logical Processes (LPs). These LPs consist of three sources, four car washes, and a single exit. They are connected as shown in Figure 3 below:

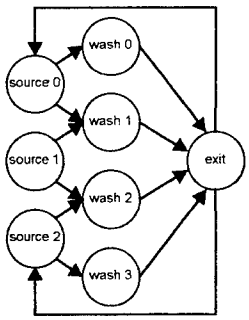

Figure 3: Car Wash Simulation

Cars are directed to the carwashes by the three sources. Each source can direct a car to one of two car washes. The source will send the car to the first idle carwash. Once a car is washed, it goes to the exit where it is inspected. If the car is not clean enough, it is sent back to either source 0 or source 2. The carwash simulation concludes when the simulation clock reaches a certain time.

The carwash exhibits several characteristics that make it useful for testing the coprocessor. It is extremely fine-grained and the LPs spend a considerable part of their time waiting for

messages. Although the fine-grained nature of the carwash simulation makes it hard to accelerate, the addition of spin-loops makes it possible to alter the simulations granularity. Since the carwash's computational demands are small, the spin loops dominate the simulation's processor time and allow accurate control of the simulation's granularity. The carwash is also interesting because it is an extremely unbalanced simulation. Since the LP named Source 1 has no inputs arcs, it never has to wait for messages from any of the other LPs. It can process events as quickly as it can schedule them in its next event queue. Source 1 does, however, schedule events on other LPs, and these LPs will have much larger next event queues to manage. The carwash simulation is also deterministic--the events are scheduled in the same order every time it is run. Normal parallel discrete event simulations are not deterministic because the simulations run on many processors and the communication latency between processors can cause events with the same time stamp to arrive in different orders each time the simulation is run. For example, if LP_1 and LP_2 schedule an event on LP_3 at the same time step in the simulation, there is no way to ensure that one message will always be processed before other. When LP_3 receives the event from the other LPs, the events will have the same time stamp so it will schedule the events in the order that they arrived. If the event from LP_1 is processed before the event from LP_2, then the simulation will probably have different results than if the events are processed in the opposite order. The carwash's determinism makes it easy to compare various simulation runs since, for all of the runs, the event should be scheduled in the same order. The determinism also makes it easy to validate the simulation's output.

2.6 *VHDL Simulations*

VHDL is a hardware description language that was developed for the United States Department of Defense as part of the very high speed integrated circuit (VHSIC) program[15:1]. Using VHDL, circuit designs can be specified at both the abstract and concrete level. A wide variety of tools have been developed to synthesize VHDL circuit descriptions into integrated circuits. VHDL simulations permit circuit designs to be tested before they are fabricated, but such simulations take a considerable amount of time for large circuits. Several theses at AFIT examined ways to accelerate VHDL simulations by running them on parallel computers. The most recent of these, by Kapp, studied the effects of a deliberate partitioning strategy for mapping the circuits to LPs[16:3-4]. Kapp's research demonstrated how an efficient partitioning strategy could minimize the dependencies between LPs and reduce the number of messages passed between nodes. Since parallel discrete event simulations tend to be communication bound, reducing the number of messages they send improves simulation performance[16:132].

AFIT's parallel VHDL simulator, VSIM, implements a subset of VHDL commands and uses SPECTRUM to synchronize the LPs. VSIM uses a commercial, sequential VHDL compiler by Intermetrics to convert VHDL into intermediate C source code[16:31]. A utility, called pbuild, converts the C source code into code compatible with VSIM. Each LP in the simulation contains a complete copy of the VHDL behaviors and SPECTRUM and this limits the maximum size of the simulation. If an LP only contained its behaviors, then larger simulations could be run and more behaviors could be simulated per node. If more behaviors were simulated on a node, the simulation's granularity and speedup would also increase. Unlike other simulations synchronized by SPECTRUM, there are two next event queues maintained for each LP. As it is in other simulations, one of the next event queues is maintained by SPECTRUM. The other next event

queue, however, is managed at the application level by VSIM. This is necessary because VSIM behaviors schedule new events by directly manipulating this queue. A filter, VHDLClocks, is used to coordinate the two next event queues and is used to implement a conservative, null message protocol based on that of Chandy and Misra.

Several VSIM simulations were used in this research. The simplest was an 8-bit, carry-lookahead adder (CLA). This circuit was relatively small, having less than 100 gates, and was useful for testing the ported versions of SPECTRUM with VSIM. The Wallace-tree multiplier simulation was also used and is much larger, consisting of over 1,000 behaviors. This multiplier multiplies two eight-bit vectors to form a twelve-bit output vector. Both the carry-lookahead adder and the Wallace-tree multiplier run for 2,000 nanoseconds. The largest VSIM simulation was the associative memory, a part of the hardware coprocessor, with over 4,000 behaviors. This memory was called an extreme search associative memory (ESAM) in Berlin's thesis and would search for a maximum or minimum value, in its contents, with O(1) complexity. The ESAM is more powerful than the Content Addressable Memory (CAM) described by Daniel since, in addition to searching for an equal value, it can search for maximum and minimum values. The ESAM's simulation size is useful because, when efficiently partitioned, it has a coarser granularity since each LP must execute more behaviors. The associative memory simulation was originally designed to run for 8000 ns, but since VSIM's clock only has enough resolution for 2000 ns, the memory's time scale was changed to picoseconds and it now runs for 8 ns. To compensate for the altered time scale, all times were scaled so that a 3 nanosecond delay became a 3 picosecond delay.

2.7 Conclusion

This chapter described the current state of research into parallel discrete event simulations. Both conservative and optimistic algorithms were examined to determine how a coprocessor could be used to accelerate them. Various hardware acceleration efforts were examined to see if any of them applied to this research, including efforts to accelerate microprocessors in general. This chapter also described SPECTRUM, the simulation environment used at AFIT, and several of the simulations that were used to test the coprocessor.

III. Porting SPECTRUM to the Intel Paragon XP/S

3.1 Introduction

The first step in implementing the coprocessor was to modify the SPECTRUM simulation environment to run on the Intel Paragon. The Intel Paragon was chosen because it is more scalable than the other computers used at AFIT and would accommodate larger numbers of LPs and coprocessors. Most of the previous parallel research at AFIT was conducted on Intel Hypercubes, and the Paragon's system calls are very similar to the Hypercube's. These similarities made porting SPECTRUM to the Paragon straightforward.

The SPECTRUM simulation environment is very similar to an operating system in that it performs communication tasks for the LPs and other tasks such as deadlock prevention. The machine-specific portion of SPECTRUM is therefore closely tied to the characteristics of the machine on which it is implemented. Before implementing SPECTRUM on a new machine, it was important to understand the architecture of the machine to which it was being ported.

3.2 Comparison of iPSC/2 Hypercube and Paragon Computers

Since the iPSC/2 and the Paragon have different architectures, it was crucial to modify the code to suit the characteristics of the machine. This type of modification involved more than just replacing new system calls in the existing program; it required looking at the overall topologies of the machines and determining how to map the simulation's functions to the nodes of the Paragon. An investigation of the machines' topologies revealed that the machines had different communication bottlenecks. These bottlenecks also played a significant role in how the

simulation environment was redesigned. One other problem involved determining what system calls were supported by both the iPSC computers and the Paragon. Since the machines were both produced by Intel and are in the same family of computers, there are many similarities between the two machines. However, some of the calls did not work on both machines. After determining how to best utilize the machine's topology, eliminate bottlenecks, and replace the machine-specific system calls, a high level analysis and design was completed.

3.2.1 Processor Capabilities

The processors used in the iPSC/2 are of a significantly different architecture than those used in the Paragon. The iPSC/2 utilizes a Complex Instruction Set Computing (CISC) Intel 80386 with a 80387 floating-point coprocessor while the Paragon incorporates a Reduced Instruction Set Computing (RISC) i860XP. A Wietek 1167 coprocessor provides additional numeric capability to the iPSC/2 [17]. The 16-MHz 80387 is rated at 0.3 MFLOPS . In contrast, the heavily pipelined, superscalar design of the i860XP processor used in the Paragon allows it to achieve a peak rating of 75 MFLOPS while only running at a clock speed of 50 MHz [18]. The floating point unit of the i860 is integrated with the CPU on the chip. One problem with the i860's floating point unit is that it lacks a floating-point divide instruction [1:167]. Instead of dividing, the i860 must take the inverse of a number using a micro-coded, iterative algorithm and then multiply the numbers. Hence, the floating-point divide performance of the i860 is much slower than other arithmetic operations. Most of the i860's built-in instructions for floating-point math do not conform to the IEEE 754 standard for floating-point math. The IEEE 754 standard has two more bits of precision for certain operations, but requires that the i860 run a math library written in assembly language[19:8-8]. This math library is much slower than the i860's intrinsic instructions.

Since the i860 gains much of its speed from pipelining, stalling the pipeline frequently will significantly slow the processor. If the processor predicts the wrong direction in a branch operation, it causes a pipeline hazard and wastes clock cycles while it refills the pipeline. The compilers used on the Paragon can optimize the code to improve the processor's branch prediction, and this had a very significant effect on the speed of computationally intense programs. Unfortunately, such optimizations are less significant on parallel discrete event simulations because of their fine granularity.

3.2.2 Machine Topologies

The Intel iPSC/2 and the Paragon are both distributed memory machines, but they have significantly different architectures. The iPSC series of computers use a hypercube topology. The hypercube topology is versatile and many different algorithms and data structures can be mapped to it with ease. The hypercube has a relatively small diameter and a large bisection bandwidth. The primary disadvantage of the hypercube topology lies in its high connectivity. The hypercube has so many connections between nodes that it is expensive to use a large number of processors. The Paragon uses a simpler topology called a two-dimensional mesh, without wraparound. The 2D mesh is simpler and scales to large sizes more easily than a hypercube, but it is not as well connected and has a smaller bisection bandwidth. The characteristics of both machines are shown in Table 2, where p is the number of processors in the system.

Characteristic	*2D Mesh without Wraparound*	*Hypercube*
Diameter	$2(\sqrt{p}-1)$	$\log(p)$
Bisection Width	$\sqrt{p}$	$p/2$
Arc Connectivity	2	$\log(p)$
Cost (Number of Links)	$2(p-\sqrt{p})$	$(p\log p)/2$

Table 2: Mesh and Hypercube Characteristics

The characteristics of the machines' topologies reveal that the hypercube has many communications advantages over the mesh. Since the mesh has a smaller bisection bandwidth, congestion is more likely to be a problem. Intel chose to address congestion on the Paragon in several ways. One technique increases the bisection bandwidth by adding more connections between nodes so that 16 bits and a parity bit can be transferred at once[20]. Another way Intel reduces congestion on the Paragon is by using wormhole routing, a variation of cut-through routing. Wormhole routing is an improvement over store and forward message passing because it does not require the intermediate nodes to store the whole message. Instead the message is forwarded by the intermediate nodes as soon as its head is received. Unlike older techniques that locked communication channels for the duration of messages, wormhole routing allows the intermediate nodes to release the connections when it receives the tail of the message.

Both the Paragon and the iPSC/2 use specialized hardware to pass messages. This hardware allows the nodes' processors to do useful work while messages are being passed and accelerates the bandwidth of the computers' communication networks. Intel claims that the Mesh Routing Component, used to pass messages on the Paragon, has a maximum throughput of 200 Mbytes per second, but the operating system overhead limits its speed to about 75 Mbytes /sec[20]. This is considerably faster than the 2.8 Mbyte/sec communication rate on the iPSC/2's

Direct-Connect Modules and implies that congestion should be less of a problem[21]. To achieve a communication speed of 75 Mbytes/sec, Intel added another i860XP processor to each node of the machine. This processor was unused until release 1.2 of the operating system, but now acts as a message processor, sharing the same memory space as the node's main processor. Figure 4 shows how this processor is connected to the node's General Purpose (GP) board:

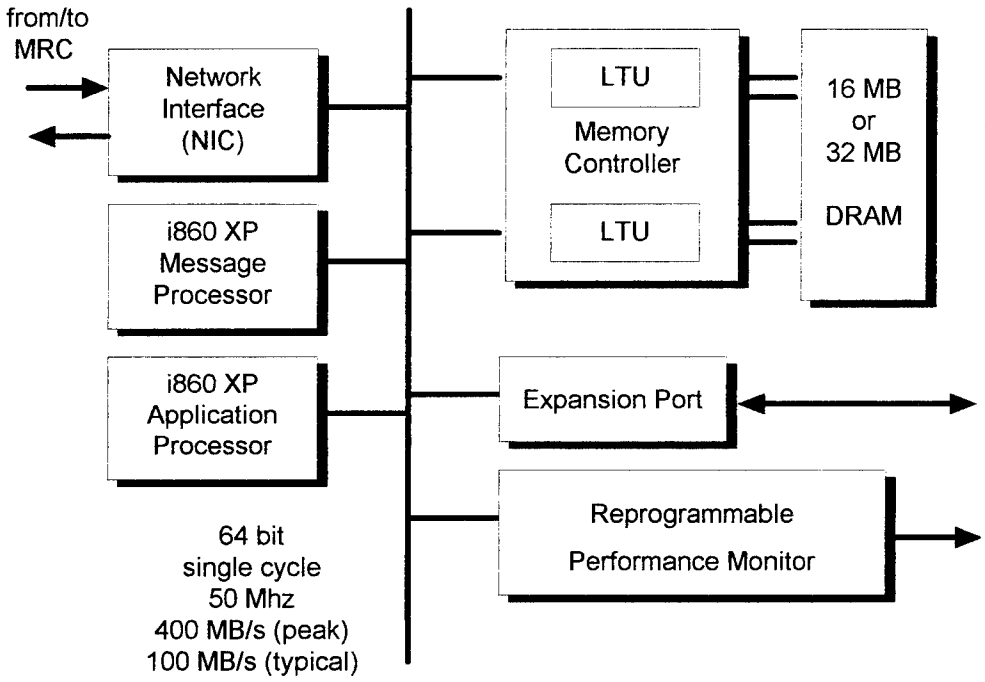

Figure 4: Intel Paragon GP Node[18]

In addition to the application and message processors, the GP node on the Paragon has many significant features. All of the parts of the GP Node are connected by a 64 bit bus with a peak bandwidth of 400 MB/sec. The Network Interface Chip (NIC) connects the board to the Mesh Routing Component and allows messages to be passed to the board via a FIFO queue big enough to hold a single packet. The memory controller contains two Line Transfer Units (LTU) and, in addition to interfacing memory to the bus, provides DRAM refresh and Direct Memory

Access (DMA) services. The expansion port can be used to add such things as memory or disk controllers. The Reprogrammable Performance Monitor (RPM) provides many services for debugging and code profiling such as the number of page faults, the number of exceptions, the amount and direction of message-passing traffic, the CPU's and memory's utilization, and a high precision timer. The RPM's timer is a 10 MHz 56-bit, global clock accurate to 100 ns local to the node and 1 μs across all of the nodes in the system[22]. From this description, it is clear that the GP node has many interesting features for applications programmers and is useful for measuring SPECTRUM's performance.

3.2.3 Operating System Differences

The Paragon and iPSC/2 both use the Unix operating system, but they use it in significantly different ways. The iPSC/2 uses AT&T Unix Version V on an external processor called the System Resource Manager (SRM), or the host node, as shown in Figure 5.

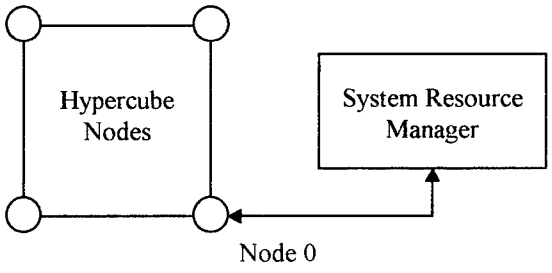

Figure 5: iPSC/2 Hypercube System Resource Manager[23:3]

The host node is typically an Intel 301 PC with a 80386 processor and is used to handle all shells and user commands. It loads programs onto the nodes and contains the language compilers and routine libraries[23]. In addition to running Unix, the iPSC/2 uses NX/2, a small, multitasking operating system on the nodes. A typical iPSC/2 program executes a host program and uses that

program to load and start the program(s) on the nodes. The host program can also pass messages to the processes on the nodes and can be used for tasks that require centralized operations.

The Paragon uses the OSF/1 Operating System, developed by the Open Software Foundation, and the Mach 3.0 microkernal on all of the system's nodes. The Paragon's version of OSF/1 Unix supports most Unix standards, such as POSIX1003.1, and has extensions to support parallel processing[24]. Some of the extensions to provide a single system image across all of the system's nodes. The single system image ". . . makes all the nodes appear to be one large system" and allows all the nodes to use a single file system[24]. This single system image, however, does not imply that all of the nodes share the same memory space; the Paragon is a distributed computer and each node has its own memory[24]. In addition to multitasking, the Paragon's operating system supports multithreaded programs and virtual memory. The NX/2 library of calls is also provided to maintain compatibility with the iPSC computers.

3.3 Paragon Communications Bottlenecks

Even though the Paragon's communication between nodes is much quicker than the iPSC/2's, there are still communication bottlenecks that significantly alter SPECTRUM's performance on the Paragon. These communications bottlenecks are not a result of slow communications within the computer, but result from poor input/output (IO) to devices outside the computer. To understand where these bottlenecks occur, it is necessary to know how the Paragon's nodes are configured and how they perform IO. A simplified diagram of the Paragon's architecture is shown in Figure 6, below. In this figure, there are several types of nodes. The largest number of nodes are the compute nodes, where the Paragon's processors are located. Many of the nodes are empty and appear to go unused. These empty nodes, however, still contain

a Mesh Routing Component (MRC), and are placed where message traffic is likely to be the highest to pass messages quickly. The service nodes are where the user interacts with the machine. When a user logs onto the Paragon, the service nodes process commands sent to the Paragon. These nodes are also where a host program would typically be run if used to coordinate activities between the nodes.

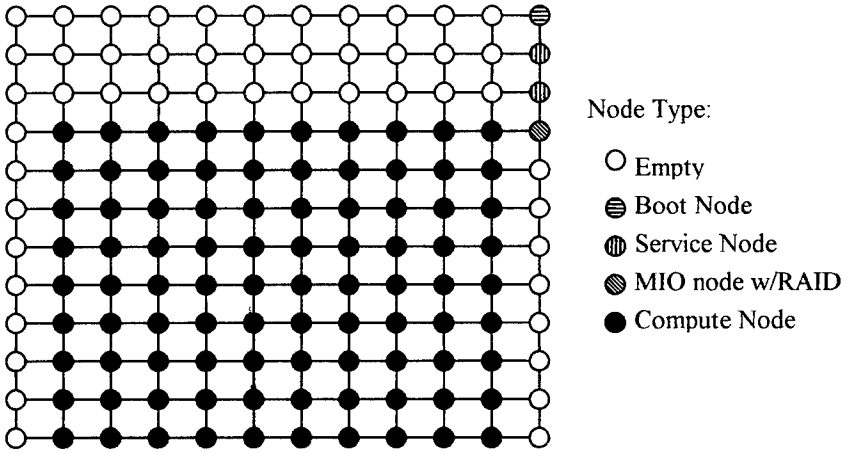

Figure 6: Intel Paragon Architecture

The Multipurpose IO (MIO) nodes can be used for several types of IO, such as to connect a Redundant Array of Inexpensive Disks (RAID) or as a network controller. The MIO card uses a SCSI interface to connect the disk drives, which is a bottleneck since the SCSI interface used in the Paragon is limited to approximately 2.4 Mbyte/sec[20]. This may not seem bad for a normal computer, but this figure is extremely low for a supercomputer which is theoretically capable of over 10 GigaFLOPS. According to Patterson and Hennessy, the overriding supercomputer IO measure is data throughput and, as a result, Intel needs to increase the throughput as it improves the Paragon's CPU performance [1:510].

Another possible bottleneck is that the Network File Server (NFS) has to pass through a MIO node or the boot node to retrieve files stored there. At Wright-Patterson Air Force Base, the boot node is the only node that connects the computer to the outside world. This boot node is likely to be a source of contention because all of the machine users will interact with the machine via this node and many of the users will try to write files to network drives on other computers that are connected to this node. One alternative to the slow IO offered by the Paragon is to use the Parallel File System (PFS). PFS still writes to the slow disks on the MIO nodes, but is does so in parallel and provides a great increase in IO throughput when several nodes can write to the disk at once.

Programs will run much faster if they avoid writing large files to disks using the normal Unix File System (UFS) and using the Network File Server (NFS). Instead, they should write to the Parallel File System or to High Performance Disk Space (HPD), which accesses the SCSI drives in the Paragon's RAIDs. It is also important to avoid writing a lot of information to the screen because this involves writing to the terminal's screen through the boot node which is very heavily used. These communication bottlenecks, and the fact that the service nodes are heavily used, led to the conclusion that the host program for SPECTRUM should not be used on the service nodes. Instead, the code was redesigned so that the host was no longer necessary. The host program for SPECTRUM, and how it was eliminated, will be discussed later in the high-level design.

3.4 System Specific Calls

The last major problem in porting SPECTRUM to the Paragon was that some of its system calls have changed from those of the iPSC series of computers. Many of the iPSC/2 calls

are still supported to maintain compatibility with older software, but since they may not be supported in future revisions of the operating system, I decided not to use them.

Another change required by the Paragon's operating system was to replace all of the references to the process id (pid) with ptype. Related iPSC commands, such as setpid( )[1] and mypid( ) were replaced by the appropriate functions, setptype( ) and myptype( ). This change was necessary because the OSF/1 Unix on the Paragon uses the pid to stand for a Unix process identifier. Each process on the Paragon has its own unique pid throughout the system and can be killed by the command: `kill pid` from any node. Since all of the processes in an application belong to the same process group, they can be killed from within the application by the command: kill(0, SIGKILL) where SIGKILL is defined in the header file, signals.h. Therefore the kill( ) command replaced the iPSC/2 command, killcube( ).

Several other machine-specific commands from the iPSC/2's host program, such as getcube( ), `startcube`, `load`, and waitall( ) were also eliminated from SPECTRUM. Since these calls existed in only the host program, and the host program was not implemented for the Paragon, these calls were no longer needed and were not replaced.

3.5 High-Level Design

The original version of SPECTRUM for the iPSC/2 consisted of four parts: cube2.c, lp_man.c, an application, and a set of filters as shown in Figure 2 in the previous chapter. The file called cube2.c is SPECTRUM's interface to the parallel computer. It contains all of the system

---

[1] In this document, all commands entered at the command line appear in this `non-proportional font`. All system function calls are denoted by their parentheses.

calls specific to the iPSC computers. To port SPECTRUM to another machine, cube2.c and its associated header file would be the only files needing changes. The file called lp_man.c is the node level manager and it performs all of the tasks of the LP. It is essentially SPECTRUM's interface to the user's application. The filters allow the functions performed by lp_man.c to be changed so that many message-passing protocols can be supported. Since the filters are the only files changed for new protocols, the user can write a simulation and not have to rewrite the simulation for every new protocol. To support optimistic protocols, however, the application would have to be changed so that it could save its state. When a causality error occurred, the application would have to restore the old state itself.

The iPSC/2 used a host program to load SPECTRUM. This host program, called getcube( ) to get a portion of the hypercube, loaded the simulation onto the hypercube, started the simulation, kept track of which processors were still running the simulation and terminated the programs on the nodes. When the host program loaded the nodes of the hypercube, it would send as many copies of the program as needed for the simulation to run. The host could send multiple copies of the program if the node needed to run more than one LP. This was possible because the nodes of the iPSC/2 could multitask. The host program could map the LPs to the nodes of the hypercubes three ways: it could read the assignment of LPs to nodes from a file, it could prompt the user to assign the LPs, and it could assign the LPs using "natural" assignment where LP 0 is mapped to node 0 and LP 1 to node 1. With natural assignment, once all of the nodes had 1 LP assigned to them, the host would then start again at 0 and continue mapping the rest of the LPs as shown in the example in Figure 7.

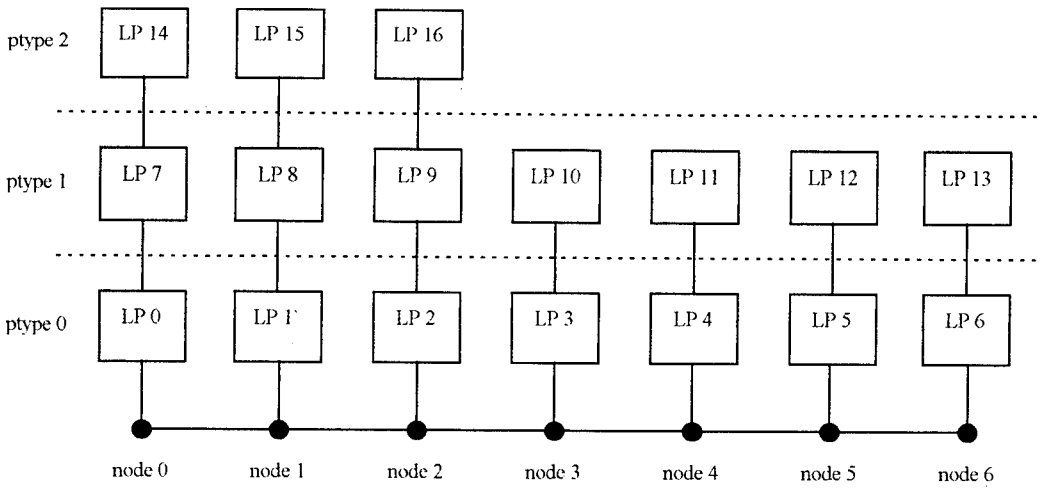

Figure 7: Natural Numbering of 17 LPs on Six Nodes

The host program coordinated the simulation shutdown by waiting for messages of type LAST_TIME. When the host program received a LAST_TIME message from an LP, it knew that the LP finished running the simulation and that it was waiting for a message of type END_MSG. When all of the LPs finished, the host program would send them a message of type END_MSG and the LPs would send the host program a message containing some information about how the simulation ran. When the host received all of the statistics from the nodes, it would execute killcube( ) to make sure that all of the nodes were terminated. The modified version of cube2.c, called mesh.c, assigns all of these host node functions to other nodes on the Paragon.

3.5.1 Redesigning SPECTRUM to Replace the Host Program

Because the Paragon version of SPECTRUM eliminated the host node, the host node's features had to be mapped to the LPs. Node_level_init( ) was altered to map the LPs to nodes in a distributed manner. To start the simulation, only one copy of the program loads per node. Each

node then calculates the number of LPs needed on that node and loads onto itself, the necessary number of LPs. The following formula determines how many LPs are needed on a node:

$$num\_lps\_needed = \left\lceil \frac{Num\_LPS - the\_node\_number}{the\_number\_of\_nodes} \right\rceil$$

Equation 2: Number of LPs Needed on a Specific Node

Once the algorithm determines how many LPs are needed, it creates the proper number of LPs per node and the LPs start the simulation. The processes then must figure out which LP they are, using the following formula:

$$lp\_number = my\_node\_number + total\_number\_of\_nodes \times my\_ptype$$

Equation 3: Determining a LP Number

The simulation, at this point, runs identically to the simulation on the iPSC/2 until a node finishes. When a node finishes, it sends a message to a control node and this node keeps track of which LPs are finished. When all of the LPs are finished, the control node sends all of the nodes an END_MSG telling them to shutdown. The nodes wait for an END_MSG before they shut down because sometimes a node sends a null message at the exact same time that the node finishes its part of the simulation. If the node did not go into a waiting state, accepting these messages, the nodes input message queue would still have a message in it and the simulation would not terminate cleanly. By waiting for an END_MSG, the nodes queues are empty and the simulation terminates cleanly. When the LPs receive an END_MSG, they send an END_STATS message to the control node and exit.

3.5.2 Changes to the Termination Algorithm

The termination algorithm also presented several challenges. The biggest challenge was writing the termination algorithm for the control node so that it would handle the following three situations:

1. The control node finishes first and has to wait for all of the other nodes to finish
2. The control node finishes last and all of the other processors have to wait for it
3. The control node finishes after some processors have finished and before others

The first two cases were relatively simple. SPECTRUM was modified to ensure that the control node was capable of receiving messages of type LAST_TIME both while it ran its part of the simulation and while it waited for other LPs to finish. The third case was a little more complicated because it required the control node to remember the state of all the other nodes across several function calls. This problem was solved by embedding the processor state in a global variable visible to all of the functions in the node level of SPECTRUM. This global variable could be updated both during and after the node ran the simulation by two different functions.

3.6 Low-Level Design

In adding the control node to SPECTRUM to replace the host program's functionality, there was a choice between using a dedicated node or adding the functions to a node also used as an LP. An examination of the control node's functions revealed that the control node was used primarily during the simulation's startup and shutdown. The only function performed while the simulation was running was keeping track of which LPs finished the simulation. Since the control

node's functions would not significantly increase the LP's load, they were placed on one of the nodes running an LP.

Many of SPECTRUM's functions had to be modified to perform the algorithms described in the high-level design. The functions that had to be modified were node_level_init( ), node_receive_pending_messages( ), node_terminate( ), shut_down( ), and node_abort( ). Three functions, all_done( ), broadcast( ), and set_control_node( ) were added.

Node_level_init( ) was changed so that it could assign LPs to nodes in a distributed manner and create enough processes using the forking algorithm described in the next section of the report. The use of the iPSC/2 command waitall( ) was eliminated from this part of the code and each LP is allowed to begin the simulation after it receives a synchronization message from the control node. The startup synchronization messages are used as follows: The control node waits for an INIT_DONE_MSG from all of the LPs after they have initialized. Once all of the LPs have sent this message, the control node broadcasts a START_SIM_MSG to them, starting the simulation. The simulation could be started without synchronization, relying on the simulation protocol to prevent causality errors, but such a simulation would be harder to instrument for performance measurements. With the synchronization, all of the processors start after the START_SIM_MSG is broadcast to them and timing measurements can be started here.

Node_receive_pending_messages( ) was extensively modified. It was altered to receive messages of type ABORT_MSG and of type LAST_TIME. The ABORT_MSG type was added so that one of the nodes could call node_abort( ) and broadcast ABORT_MSGs to all the other nodes, telling them to abort the simulation. The LAST_TIME message type was added so that the control node could receive them and tell when an LP had reached its last simulation time. If the

LAST_TIME message is sent to any other node in the simulation besides the control node, the simulation is aborted.

Node_abort( ) was also modified extensively. It previously sent messages to the host of the iPSC/2 to tell it to end the simulation. Since the host was eliminated, node_abort( ) was modified to broadcast ABORT_MSGs to all of the LPs using the new function, broadcast( ). Node_abort( ) was also modified to detect whether or not the simulation had completely initialized. If the simulation had not initialized when node_abort( ) was called, then it would try to abort nodes that didn't exist. To fix this problem, node_abort( ) uses kill( ) to kill all the processes if the simulation has not initialized. The global variable, init_not_done, was added so that the LP would know if initialization was completed when it was set to 0. The variable, init_not_done, is initialized to 1 and is set in two places. The control node sets it to 0 when it receives INIT_DONE_MSGs from all of the other LPs. The other LPs set init_not_done to 0 when they receive a START_SIM_MSG from the control node. Both of these assignments to init_not_done occur in node_level_init( ).

Node_terminate( ) was changed so that it had two routines embedded in it. One routine was similar to the iPSC/2 implementation, and the second routine was added so that the control node could shut down all of the other nodes. The second routine originally used Intel's broadcast to send all of the other nodes the END_MSG, but this had to be changed for the cases when SPECTRUM uses a different number of LPs on a node. Intel's broadcast would send messages to all of the nodes whether or not they actually had a process of that ptype. If a node did not have a process of the Intel broadcast's ptype, then the message would wait in the node's input buffer and cause the simulation to hang when it tried to shutdown. The broadcast( ) function was added to fix this problem. Node_terminate( ) was also changed to call another new function, all_done( ), to determine when the simulation was finished on all of the nodes.

This research added all_done( ) to the file mesh.c, designing it to look at an array of processor states and determine which processors had not finished. The processor states were added to the global structure called lp_table by adding a new field called LP_state. If LP_state equals 0 then the LP is still running the simulation. If it is 1, the LP has finished. If all of the LPs are finished, all_done returns a 0. If the LP has not received a message of type LAST_TIME from all of the nodes, then all_done returns a 1.

The function broadcast( ) was added to simplify broadcasts to all the processes running the simulation. It was written because of the previously described problems with Intel's broadcast and because Intel's broadcast only sent messages to a single ptype. If the simulation was running more than one LP per node, the Intel broadcast would have to be called once for each ptype. Other global commands such as gsync( ) work in the same manner as Intel's broadcast and had to be avoided. The syntax of Intel's broadcast is: csend(type, message, length, -1, ptype) where the -1 in the node parameter indicates that the message is to be broadcast. The new broadcast has the following syntax: broadcast(type, message, length). It first determines the number of ptypes used by every node and the number of ptypes only used on some of the nodes. Then it uses Intel's broadcast to send the message to processes that have a ptype used on every node. Intel's broadcast is used at this point because it uses a binary tree structure to broadcast the message more quickly than if individual messages were sent. The rest of the processes have a ptype that is used on only some of the nodes and are sent individual messages.

Another function, set_control_node( ), was added to insure that the control node was set properly. The Paragon version of SPECTRUM stores the location of the control node in the variables cnode and cptype. These variables should only be altered by set_control_node( ) so that it can check to ensure they point to a node that actually exists. This function is called by node_level_init( ), which passes it the variables that identify the control node's number and ptype.

These variables are currently fixed to make node 0, ptype 0 the control node since this node will always be used for any simulation.

When these algorithms were redesigned, this effort made them as robust as possible by checking for errors that would be impossible under the normal operation of the program. One such error was that some of the data structures used pointers and an invalid pointer could cause a segmentation fault. The routine that updated the processor state when the control node received a message of type LAST_TIME caused the most concern. This routine stripped an integer from the LAST_TIME message and used that integer to calculate a pointer into an array. If the message was scrambled, and the integer stripped from it was wrong, the resulting integer could cause a segment fault when it was used as the array's index. To increase SPECTRUM's robustness, bounds checking was added to this routine and several other places where invalid pointers might cause segmentation faults.

3.7 Implementation

The implementation of SPECTRUM on the Paragon was relatively straightforward except for the algorithm used to fork new LPs on a node. The data structures required few modifications, but some of the functions had to be modified extensively.

3.7.1 The Use of fork( )

Because it is fast and memory efficient, fork( ) was used to create the multiple processes on a node and assign them their new ptype numbers. When a node forks, it does not copy the entire code of the new process. Instead, it uses reentrant code and has two data spaces, one for each thread of control. All of the variables that have the same value are shared between the two programs and they are only copied when one of the processes writes a new value to them. This

works to SPECTRUM's advantage because the only values that are defined when the program forks are values that are constant among the processes. In fact, forking not only eliminates the need to load another program onto the node, it eliminates the need for each LP to load the arc file describing the arrangement of the simulation's communication arcs. With forking, only the first LP on a node reads the file and the information is common to all of that node's children that it forks. The fork algorithm is shown in Figure 8 and consists of a loop that uses the array of integers, lp_num_table, to hold the number of times that the LP needs to fork. The numbers in

```
for(i=0; i < (lp_num_table[mynode()] - 1); i++)
      {
        if ((pid = fork()) < 0)
          nx_perror("fork error in node_level_init");
       else if (pid == 0)
         {           /* then child sets ptype and continues loop */
          setptype(i + 1);
          printf("forked on node(%ld) ptype(%ld)\n", mynode(), myptype());
         } else      /* parent let the child do the rest of the forking */
          break;
      }
```

Figure 8: SPECTRUM's fork algorithm

lp_num_table are calculated using Equation 3, described previously. The forking algorithm worked successfully and allowed me to assign any number of LPs to a node.

3.8 Capabilities That Were Not Implemented

Due to time limitations, this research was not able to make all of the changes originally intended for SPECTRUM. One such change was the ability to place comments within the arc files. Another capability not added was the ability to map the LPs to nodes from a file. This

capability would be very useful and would allow the simulation programmer more flexibility on how the simulation was partitioned. It could be added to node_level_init( ) on the control node when it starts the simulation by broadcasting the START_SIM_MSG. The control node could broadcast the necessary information to the other nodes in the system so that they would know how many times to fork and how to assign LPs to the new processes. While these changes would be useful additions to SPECTRUM, their absence does not limit its utility.

3.9 Conclusion

Porting SPECTRUM to the Intel Paragon entailed tailoring it to the Paragon's architecture. The systems' support of the same message passing calls greatly simplified the move from the iPSC/2 to the Paragon. The host node, used on the iPSC/2, was eliminated and the host node's functions were placed on the control node, which also was one of the LPs. The files mesh.c and mesh.h replaced cube2.c and cube2.h, respectively. Several functions in the file mesh.c had to be modified and three new functions were added. Most of these changes involved adding the control node's functions, but some of them also implemented a distributed mapping algorithm used to assign nodes to LPs. The fork( ) command was used to create the processes needed for the required number of LPs on the nodes. Once these changes were complete, the simulations' make files were altered and SPECTRUM was tested on the Paragon. The simulations' performance results on the Paragon are described in Chapter 6.

IV. Software Architecture of the Coprocessor

4.1 Introduction

The decision to use a general purpose microprocessor changed the nature of this research. Instead of being a hardware design effort, the coprocessor became a software project where the parallel computer would be use some nodes to run logical processes and others to run the coprocessor. To be useful, the coprocessor would need to exploit parallelism in the tasks performed by an LP. From a high-level perspective, an LP performs four basic tasks:

1. An LP executes events (NEQ).
2. It schedules new events in its NEQ.
3. It passes events between itself and other LPs.
4. It synchronizes itself with other LPs to prevent causality errors.

The only application-specific task in the above list is when an LP executes an event. All of the other tasks are generally handled by the simulation's environment. The simulation environment is used to schedule events because the events may need to be scheduled on another LP's NEQ. It is also important to note that passing events and LP synchronization are closely related, especially for conservative protocols that wait for events from other LPs.

The coprocessor was designed to minimize work performed by the node running the LP. More specifically, it was designed to manage the LP's NEQ, handle message-passing between LPs, and perform synchronization tasks. Before these tasks could be offloaded to the CP, SPECTRUM's flow of control, functions, and global variables had to be analyzed to see how they would be affected by running on a node other than the node running the application. After

SPECTRUM was analyzed, it was modified to support a coprocessor and renamed CPSPECTRUM.

4.1.1 SPECTRUM's Layers Revisited

SPECTRUM's layered architecture, shown in Figure 2, simplified the coprocessor's interface to the application software because it limited the functions that the LP called. An application is only supposed to call SPECTRUM's functions in the LP Manager layer. The only other function, outside of the LP Manager layer, that an LP might call is node_abort( ). By moving the bodies of these functions to the coprocessors and replacing them with small functions of the same name, it was not necessary to modify the simulations to run using the CPs. The replacement functions pass a message to the CP, requesting that one of SPECTRUM's functions be run and, if necessary, wait for a reply from the CP. The following functions can be called by an LP: lp_level_init( ), lp_init( ), lp_post_event( ), lp_get_event( ), lp_advance_time( ), lp_terminate( ), and node_abort( ). The LP can also call the following utility functions: read_lp_info( ), display_lp_info( ), open_file( ), errlog( ), display_event_q( ) and log( ). Implementing these functions on a coprocessor would place all of the NEQ management tasks, message passing tasks, and synchronization tasks on the coprocessor and would minimize the LP's load. SPECTRUM required extensive changes to support such a coprocessor, but these changes did not change SPECTRUM's high-level behavior. Because many simulations are already designed to run on SPECTRUM, backwards compatibility was a key goal in CPSPECTRUM's design. Since the filters interact frequently with the LP manager functions, they had to be moved to the coprocessor as well. Moving the filters presented several problems because they can make calls to the application layer. Before any of these changes were implemented, SPECTRUM's flow of control had to be analyzed.

4.2 High-Level Analysis of SPECTRUM's Functions

Moving SPECTRUM's functions and filters to the coprocessor required a thorough understanding of SPECTRUM's functions and variables. Since SPECTRUM's layers are implemented in several files, and these files do not explicitly state the location of the functions that they are calling, it was very difficult to determine the order in which SPECTRUM called functions. To follow SPECTRUM's flow of control, several diagrams, similar to flow charts, were made that showed how SPECTRUM's functions were called. An example diagram is shown in Figure 9.

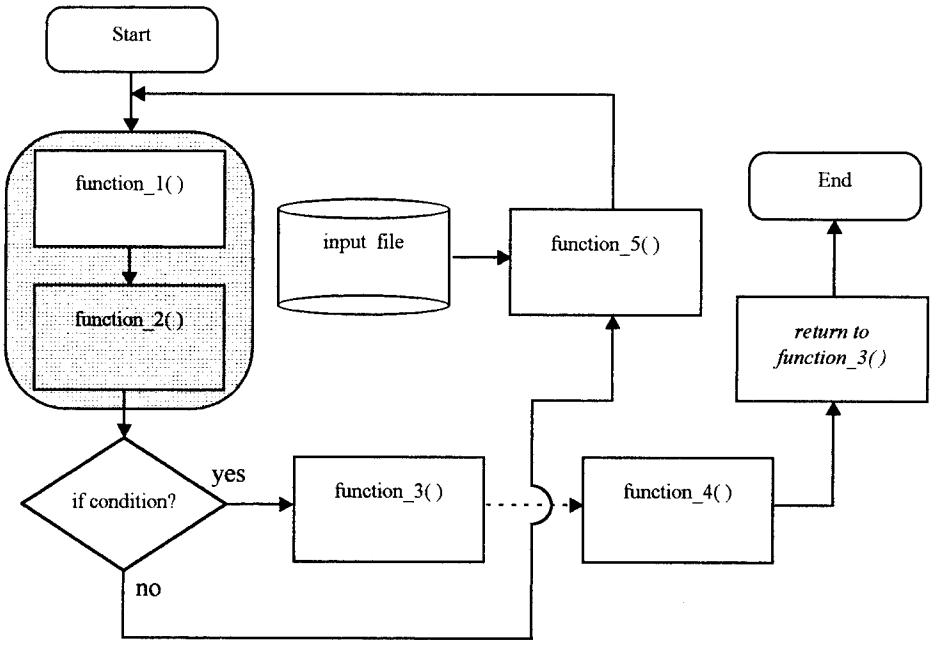

Figure 9: Example of Flow Control Diagram

In this figure, each rectangle represents a function call and each diamond represents a branch that determines what functions will be called. The program in Figure 9 begins with the block labeled "start" and immediately calls function_1( ). The gray box with rounded edges denotes that function_1( ) calls function_2( ) and that function_2( ) returns before function_1( )

exits. Gray boxes can be nested and that the function calling the other functions in the gray box is also within the gray box but is not gray. The diamond, as mentioned before, represents a branch that affects the program's flow of control. If the condition specified in the diamond is true, then function_3( ) is called, otherwise function_5( ) is called. It is important to note that the diamond represents only a decision that affects the program's flow of control. The program may have many other conditional branches, but they are not shown if they do not affect which functions are called. If function_5( ) is called, then the program starts over at function_1( ). The conditional construct shown in this diagram therefore implements a loop. Depending on the diamond's placement, many types of loops, such as "for" loops or "while" loops may be constructed. The dashed line between function_3( ) and function_4( ) means that function_3( ) calls function_4( ) and function_4( ) does not return to function_3( ) immediately. Instead, function_4( ) returns the flow of control to function_3( ) later, at the box labeled: *return to function_3( )*. Boxes with labels in italics can also be used to denote other important events that may not affect the flow of control. The disk-shaped box labeled "input file" represents a file read by function_5( ). If the arrow was pointing the other direction, it would represent a file being written.

These flow of control diagrams provided useful insights into how SPECTRUM works and helped resolve many problems during the coprocessor's implementation. A diagram was completed for each of the functions in SPECTRUM's LP Manager layer and shows how these functions call filters and functions from SPECTRUM's Node Manager layer. Another diagram, shown in Figure 15, illustrates the flow of control for a typical SPECTRUM-based application.

4.2.1 LP Manager Functions

A SPECTRUM-based simulation interacts with other nodes through the LP Manager layer. The application layer typically uses the following functions from this layer:

lp_level_init( ), lp_init( ), lp_post_event( ), lp_get_event( ), lp_advance_time( ), and lp_terminate( ). These functions also call several other functions in the LP Manager layer such as lp_post_message( ) and lp_nq_event( ). Before implementing the coprocessor, each of these functions were examined and their roles in the simulation were determined.

An application's first call to SPECTRUM is always to lp_level_init( ) which initializes SPECTRUM and the filters. The application passes an array of pointers to functions and an array of arguments to lp_level_init( ). These pointers are set to point to functions that run an LP, and the application passes the arguments to these functions when the LP starts. For example, in the carwash simulation, there are eight LPs. Each LP has its own function that describes how it runs. In the VHDL simulations that use VSIM, each LP runs the same function, but different arguments tell each LP which of the gates it is supposed to simulate. This arrangement of LPs is versatile because each LP-node can run any of the LPs, but is inefficient because the LPs have unnecessary code from the other LPs that is not executed and wastes memory.

When lp_level_init( ) is called, it reads an arcs file that describes the number and the qualities of communication paths between LPs. The arc delays are also read from the arcs file. With SPECTRUM, the number of communication arcs and their delays are static. After the arcs file is read, node_level_init( ) is called. Since node_level_init( ) is a node manager function, it will be described later. Node_level_init( ) calls the appropriate LP function in the application layer and starts the application. The application's LP function then calls lp_init( ), which initializes the synchronization protocol's filter. SPECTRUM's initialization up to this point is depicted in Figure 15.

Once lp_init( ) sets up the application's filters, the first event in the simulation is scheduled using the function, lp_post_event( ). Whenever an LP needs to schedule an event, it

can schedule it using lp_post_event( ), even if the event will be executed on another LP. SPECTRUM's events have the following structure:

```
typedef struct event {
   int from_lp;    /* lpid of lp sending event */
   int to_lp;      /* lpid of destination lp */
   int time;       /* timestamp of event */
   int event;      /* event type or number */
   int id;         /* signal id */
   int value;      /* signal value */
   int aux;        /* to be used as needed by application */
   int line_num;   /* some lp pairs may have more than 1 line */
   int bufsize;    /* Sizeof following databuf */
   char *databuf; /* Pointer to user provided buffer */
   struct event *next;
};
```

Figure 10: SPECTRUM's Event Structure

Most of the event's fields are self explanatory. The to_lp and from_lp fields describe which LPs the event is to and from. The time field is the time at which the event occurs and is used by both the synchronization mechanisms and the next event queue. The event field is used to determine the event's type and is application specific. If a simulation uses null messages, it sets the event field to the value of a simulation-defined constant. The rest of the fields are also application specific. The databuf field allows information to be packed into an event and sent to another node. The databuf can be used for such tasks as load balancing, where the information sent would not fit into any of the other fields. The bufsize is the length, in bytes, of the databuf and must be set correctly for the databuf to be passed. The next field is a pointer to another event and is used by the NEQ to link the events stored in it. If an event has not been placed in the NEQ, this field is set to null.

A flow of control diagram for lp_post_event( ) is shown in Figure 11. This flow of control diagram illustrates many important facts about how SPECTRUM's LP Manager works. The first task performed by lp_post_event( ) is to call the POST_FILTER, if one is present. The filter can use the information in the event's structure to determine what to do with the event. If the filter determines that the message is unnecessary, it will replace the pointer to the event with a null pointer and lp_post_event( ) will return without sending the event. If the event is not eliminated by the filter, lp_post_event( ) then determines whether or not it needs to be sent to other nodes. If the event occurs on another LP, it is sent to that LP via node_send_message( ). If the event is to be scheduled on the LP's NEQ, then a new event is created and that event is passed to lp_post_message( ). The new event is created, using node_create_event( ), so that the event can be placed in and removed from the queue independently, without the application freeing its memory. Since lp_post_event( ) copies the event to a new event, the application is expected to free the event that it passes. The new copy of the event is passed to lp_post_message( ) where it is scheduled on the LP's NEQ.

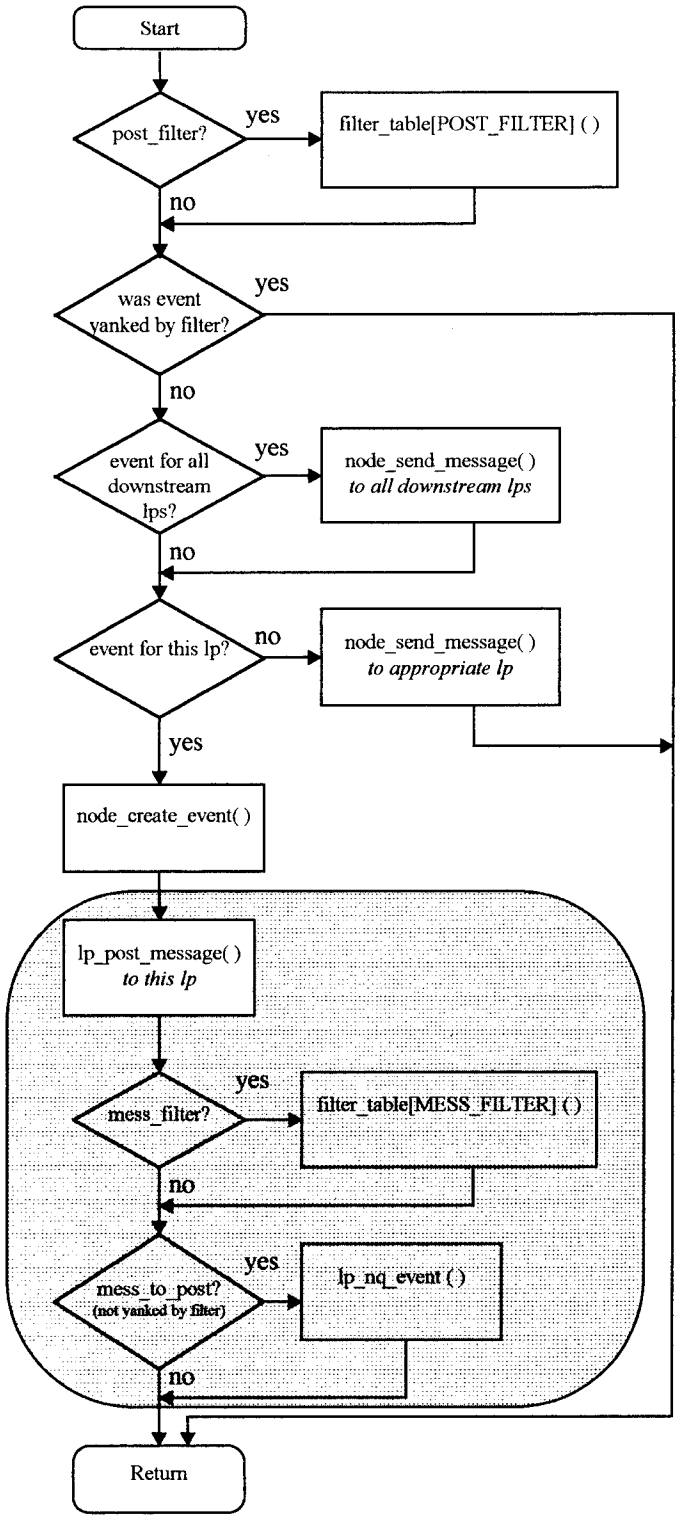

Figure 11: lp_post_event( )'s Flow of Control

The flow of control diagram for lp_post_message( ) is shown in Figure 12. Like lp_post_message, it receives an event and calls a filter before it does anything to the event. The filter for lp_post_message( ) is called MESS_FILTER. After the filter has been called, the function checks to see if the event eliminated the message. If it did, the function returns. If the event was not eliminated, the event is placed on the NEQ by lp_nq_event( ). Even though SPECTRUM uses lp_post_message( ) to put an event in an LP's NEQ, it is important that the application does not use it in the same way. The application should always use lp_post_event( ) to place a message on the queue because using lp_post_message( ) would bypass the POST_FILTER. Even though the POST_FILTER would not eliminate an event scheduled to run on its LP, it could use the event's time stamp to update some of the synchronization protocol's internal structures.

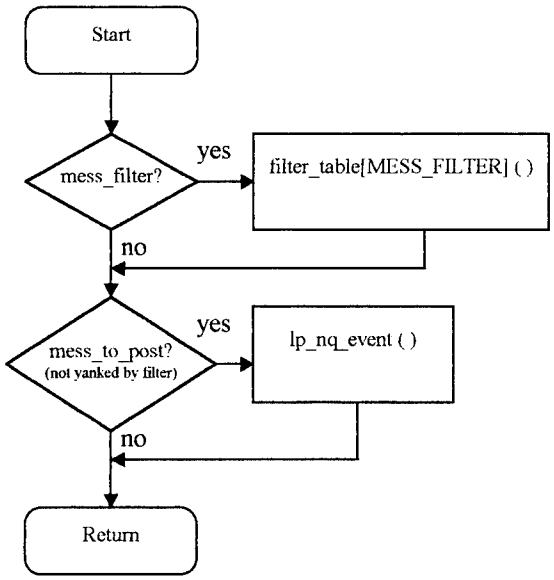

Figure 12: lp_post_message( )'s Flow of Control

SPECTRUM uses the function, lp_get_event( ) to get an event from the next event queue. The flow of control diagram for lp_get_event( ) is shown in Figure 13. The application does not pass any parameters to lp_get_event( ) but receives an event from it when it returns. The first task performed by lp_get_event( ) is to call the filter, GET_FILTER. If the filter is present, it is used to receive the event and the rest of the function is bypassed. If the filter is not used, then lp_get_event( ) calls node_receive_pending_messages( ) to see if any messages are waiting. If no messages are waiting and there is at least one event in the NEQ, lp_get_event( ) returns the top event in the queue. If there are messages waiting, lp_get_event( )'s first call to node_receive_pending_messages( ) will receive up to MAX_MESSAGES of them, where MAX_MESSAGES is a constant defined in SPECTRUM's node layer. If no messages are waiting and the NEQ is empty, lp_get_event( ) calls node_block_til_message( ) which waits for a message to be received. When a LP waits for an event or a message from another node, it is said to be blocking. Blocking is an overhead resulting from the parallel implementation. Its severity depends on the simulation's synchronization protocol and the amount of lookahead the protocol is able to exploit.

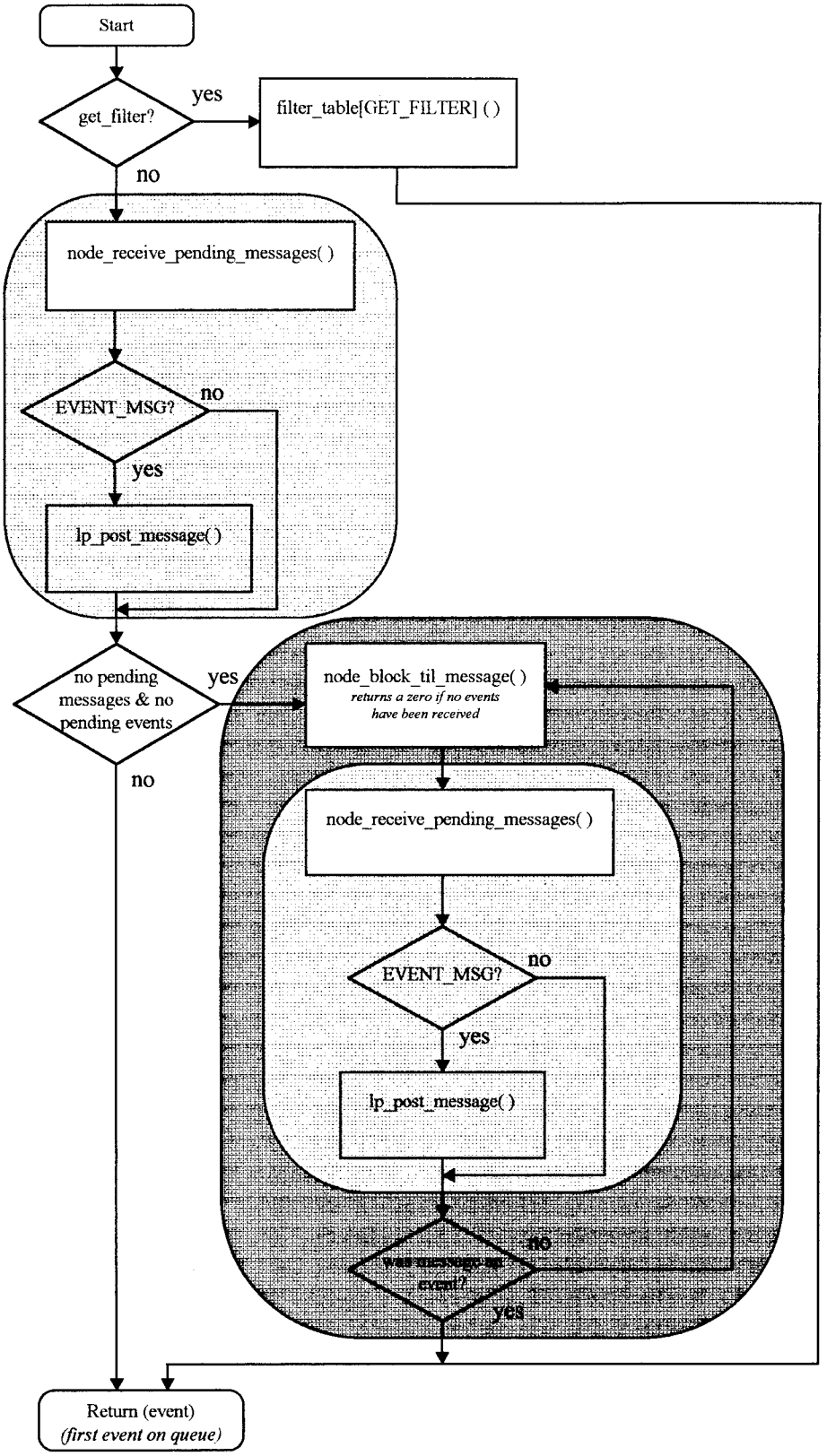

Figure 13: lp_get_event( )'s Flow of Control

Since lp_get_event( )'s functionality can be entirely superseded by the filter's, it is versatile and can implement a wide range of simulation protocols. This versatility is evident in the way VSIM was implemented. Because VSIM uses a NEQ at the application level in addition to the NEQ built into SPECTRUM, its GET_FILTER must be able to look at both queues. If VSIM's queue is ahead of SPECTRUM's, VSIM must still call lp_get_event( ) to make sure that executing the event at the top of its application-level queue will not cause a causality error. In the case that SPECTRUM's next event occurs after VSIM's, lp_get_event( ) should not return its earliest event. Instead, the filter checks the input arcs, decides which queue has the earliest event and returns a null pointer if VSIM's queue is ahead. Before it returns this null pointer, it must wait for any synchronization messages necessary for the LP's next event.

To advance the simulation clock, the application calls lp_advance_time( ). This function uses a filter called TIME_FILTER to make sure that the time can be advanced. The application passes the function a time, and the filter makes sure that the new time is valid. If the time requested is not valid, the filter must handle the error. The application usually calls lp_get_event( ) to get an event, and then uses the time stamp from that event in the call to lp_advance_time( ). SPECTRUM uses the global variable, my_clock, to hold each LP's simulation time and uses lp_advance_time( ) to update this variable. The flow control diagram for lp_advance_time( ) is shown in Figure 14.

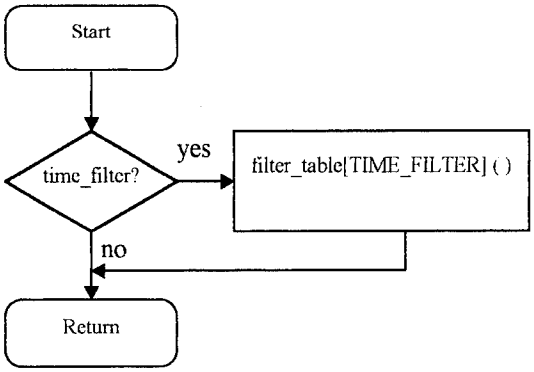

Figure 14: lp_advance_time( )'s Flow of Control

When an LP's local clock is greater than or equal to the time at which the simulation is supposed to end, it calls lp_terminate( ). This function calls the TERM_FILTER, to inform the filters that the simulation is about to end, and then calls node_terminate( ), which ends the simulation. This function's flow of control may be seen in Figure 15.

Simulations access SPECTRUM's features using the functions: lp_level_init( ), lp_init( ), lp_post_event( ), lp_get_event( ), lp_advance_time( ) and lp_terminate( ). The following table summarizes their use:

Function	Input Parameters	Returns	Purpose
lp_level_init( )	LP functions, args		Starts SPECTRUM
lp_init( )	LP #		Initializes filters
lp_post_event( )	Event		Posts an event to one of the LP's NEQ
lp_get_event( )		Event	Gets first event from NEQ, receives messages from other nodes
lp_advance_time( )	The New Time	[1]	Advances SPECTRUM's simulation clock
lp_terminate( )	Event		Removes filters so simulation can terminate

Table 3: Summary of SPECTRUM's Functions

[1] lp_advance_time( ) does not return a value, but it does have the side-effect of updating my_clock.

The functions in Table 3 are important in the design of the coprocessor because they are the only functions, aside from node_abort( ), that the coprocessor will have to support. When the application makes a call to one of these functions, the node running the LP will have to send a message to the CP telling it to perform one of SPECTRUM's functions. If the function returns a value, then the LP will have to wait for the CP to pass a message, containing the returned value, back to it.

4.2.2 Node Manager Layer

Most of changes necessary to implement the coprocessor were made in the Node Manager Layer. The coprocessor needs this layer to initialize itself, send and receive messages, and terminate the simulation. Before adding these capabilities, SPECTRUM's existing capabilities had to be examined.

The first function that SPECTRUM calls in the node manager layer is node_level_init( ). This function initializes all of the global variables used to map the LPs locations and communications arcs. It also determines how LPs are mapped to nodes and synchronizes the LPs as they start up. The LPs start when node_level_init( ) calls their functions, and can be timed during this call to see how long they run.

The heart of the Node Manager Layer consists of the functions used to send and receive messages from other LPs. The function, node_receive_pending_messages( ) is used to receive all messages sent to the LP while it is running. Node_receive_pending_messages( ) starts by checking to see if any messages have been received by the node and are in the node's message buffer. If a message is waiting in the buffer, node_receive_pending_messages( ) will receive the message, examine its type, and process it. If a message is not waiting, node_receive_pending_messages( ) will return a zero. Another function, called

node_block_til_message( ), is used to wait for a message to be placed in the nodes message buffer by the operating system. It calls node_receive_pending_messages( ) to receive the message. If the LP is blocked and is waiting for an event, node_block_til_message( ) will call node_receive_pending_messages( ) until an event arrives that will allow the simulation to continue. Node_block_til_message( ) blocks when waiting for a message by using a synchronous probe. When an event is received by node_receive_pending_messages( ), it calls lp_post_event( ) to post the event to SPECTRUM's NEQ. Next, lp_post_event( ) calls lp_post_message( ) which uses the MESS_FILTER to determine if the next event can be executed. The MESS_FILTER will be described in the next section. The MESS_FILTER or lp_post_message sets one of two flags to inform node_receive_pending_messages( ) that the next event can be executed. The first flag, called posted, is set by lp_post_message( ) when it posts an event to the NEQ. The second flag, processed, was added by Hurford to allow an application to try to execute the next event after it receives a null message[25]. Normally SPECTRUM would need to wait for a real event before continuing, but if the application has its own queue, the updated arcs from the null message may allow it to execute an event from its application level queue. The processed flag is used by VSIM's filters in the file, vhdlclocks.c. If neither the posted or the processed flags are set, node_receive_pending_messages( ) returns a zero and node_block_til_message( ) will wait for another message to be received.

The coprocessor needs to handle additional message types such as LP requests, and node_receive_pending_messages( ) is the ideal place to add this capability. It is called frequently and it will be able to handle requests from the LP node quickly. Another advantage of receiving the LPs' requests for SPECTRUM services in node_receive_pending_messages( ) is that while the CP is waiting for a request, this same function can be receiving and enqueueing events from other

LPs. Most of the parallelism the coprocessor exploits results from its ability to receive and enqueue messages while its LP is processing events.

The node level manager uses the function, node_send_message( ) to send a message to another processor. This function, which is passed an event, examines the event's to_lp field and uses some of the global mapping variables to determine which node to send the message. Since node_send_message( ) can only be passed an event, any other type of message will have to be embedded in an event. The global mapping variable that this function uses to determine the message's destination had to be changed to send the message to the coprocessor, but other than that, this function was unchanged.

4.2.3 Filter Functions

SPECTRUM's filters are called by the functions in the LP manager layer. When lp_level_init( ) is called, it calls a function, build_table( ), which is supplied by the filter's code. This function constructs a table of pointers pointing to the filter functions that the LP manager will call. The pointers are stored in an array, called filter_table, which is indexed by five constants: INIT_FILTER, GET_FILTER, POST_FILTER, TIME_FILTER, MESS_FILTER, TERM_FILTER. The following table summarizes which filters are used by the LP manager's functions:

Function	Filter Table Index
lp_level_init( )	None
lp_init( )	INIT_FILTER
lp_post_event( )	POST_FILTER
lp_post_message( )[2]	MESS_FILTER
lp_get_event( )	GET_FILTER
lp_advance_time( )	TIME_FILTER
lp_terminate( )	TERM_FILTER

Table 4: SPECTRUM's Filter Pointers

Since the filters are tied closely to both the application and the LP manager layer, the filters can make calls to either layer. Most of the filters' calls are to the LP manager and they use many global variables defined by SPECTRUM. Because of these function calls and variables, the filters run on the coprocessor.

4.2.4 A Typical SPECTRUM Simulation

Figure 15 is a flow of control diagram from a typical SPECTRUM-based application. This particular diagram is based on the carwash simulation and reveals many insights into how SPECTRUM's functions are used by the application. One useful section of the diagram, labeled "Main Simulation Loop," shows how the application uses lp_get_event( ), lp_advance_time( ), and lp_post_event( ) to get events, advance the simulation clock and schedule new events.

---

[2] lp_post_message is not called directly by the application, but is called by lp_post_event( )

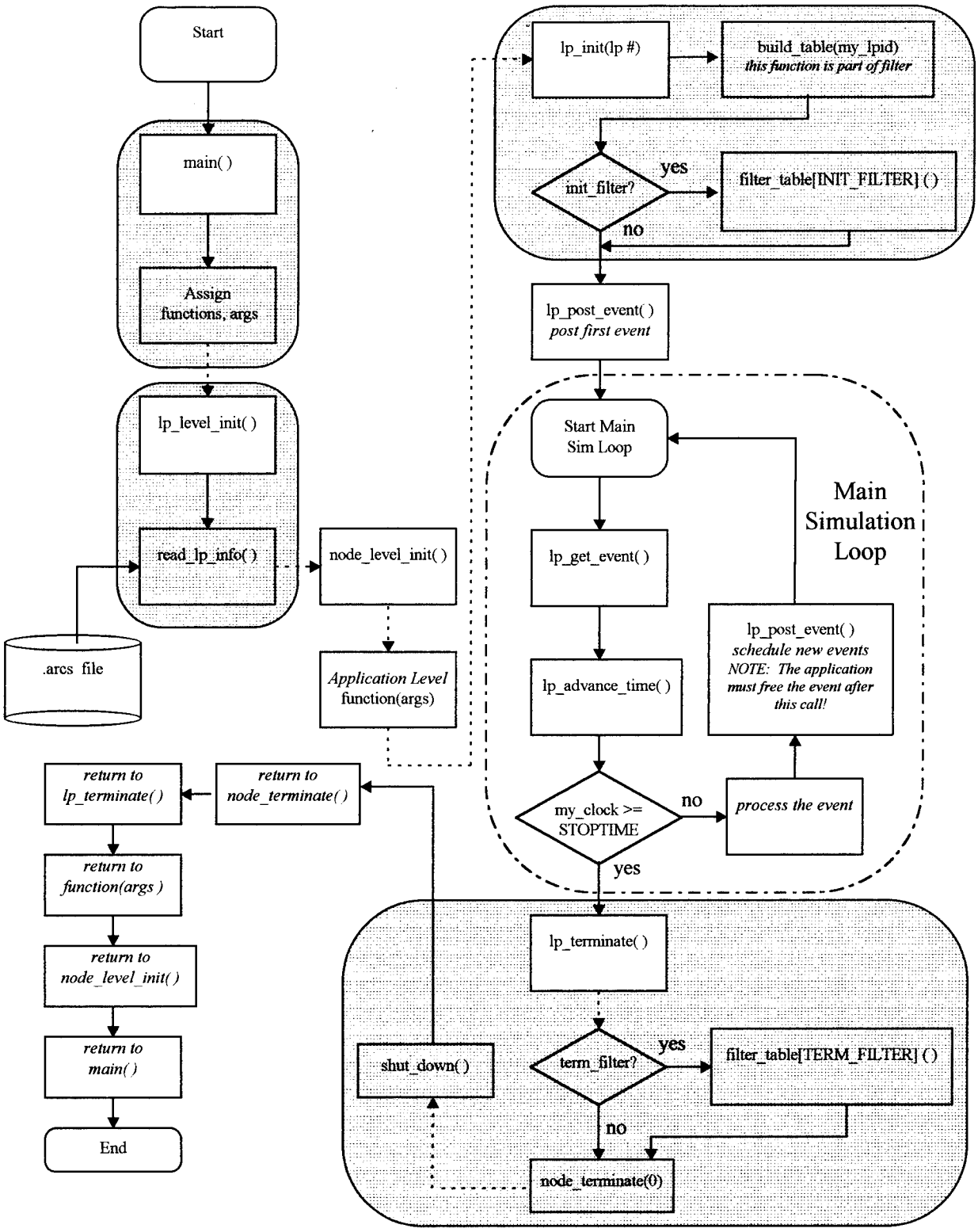

Figure 15: SPECTRUM High Level Flow of Control for a Typical Application

4.3 Low Level Design

The coprocessor's design reflects two main goals: to maintain compatibility with existing simulations and to minimize the work performed by the nodes running the LPs. The detailed analysis of SPECTRUM in the previous section of this report was very useful for accomplishing the first goal. Moving the NEQ management functions and synchronization tasks to the CP accomplished the second. A third goal was to minimize the communication latency added by the use of the coprocessor. Unfortunately, since the coprocessor is on a separate node from the LP, it was difficult to complete the third goal. The only possible way to minimize the communication latencies between the LPs and the CPs was to insure that they were located on nodes that were close together. Because most of the Paragon's communication latency is the time that the operating system takes to format the messages, placing the LPs and CPs close together did not significantly reduce their communication latency. Placing them near one another, however, did minimize congestion and reduced communication latency when large numbers of nodes were used.

4.3.1 Coprocessor Topology

With the coprocessor, LPs and CPs are mapped in a similar manner to natural order. The main difference from SPECTRUM's usual natural ordering is that coprocessors are placed between the nodes running the LPs. To maximize the coprocessor's time on its node's processor, only one coprocessor can be mapped to a CP node. Since the LP will communicate frequently with its CP, the CP nodes are located between the LPs, thus minimizing contention. The CPs are also mapped in natural order, with as many CPs between the LP-nodes as there are LPs on the previous LP-node. An example of this mapping is shown in Figure 16 where the arrows denote which LP uses which CP.

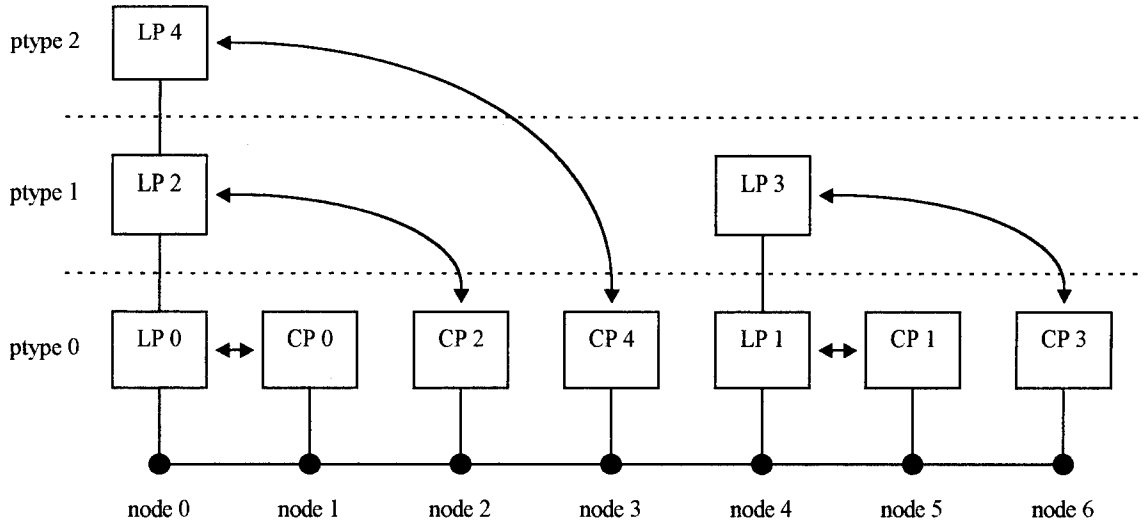

Figure 16: Example of Coprocessor Topology Using 6 Nodes and 5 Logical Processes

With this mapping scheme, the simulations using the coprocessor need at least $n+1$ nodes to run, where n is the number of LPs. With the minimum number of nodes, one node will run n LPs and the other nodes will run 1 coprocessor each. The maximum number of nodes that the simulation can use with the coprocessor is $2n$, where n is the number of LPs. If the maximum number of nodes is used, there will be an LP on every even numbered node (assuming 0 is even) and a CP on every odd numbered node. Ideally, there would be only one LP per node running LPs. For all of the tested simulations, the Paragon was large enough to use the ideal number of nodes. The coprocessor was also tested extensively with fewer than ideal nodes to insure that larger simulations could be run.

4.3.2 CPSPECTRUM's Layers

CPSPECTRUM's architecture had to be expanded to support the coprocessor. Several new layers needed to be added so that the application could use SPECTRUM's original interface and to enable the filters to obtain information from the application. These layers are shown in

Figure 17. The nodes running LPs have two layers: the Application Layer and the LP Interface Layer. The Application Layer is the same as the original version of SPECTRUM. The LP interface layer allows the application to make calls to SPECTRUM's LP manager by passing messages to the CP.

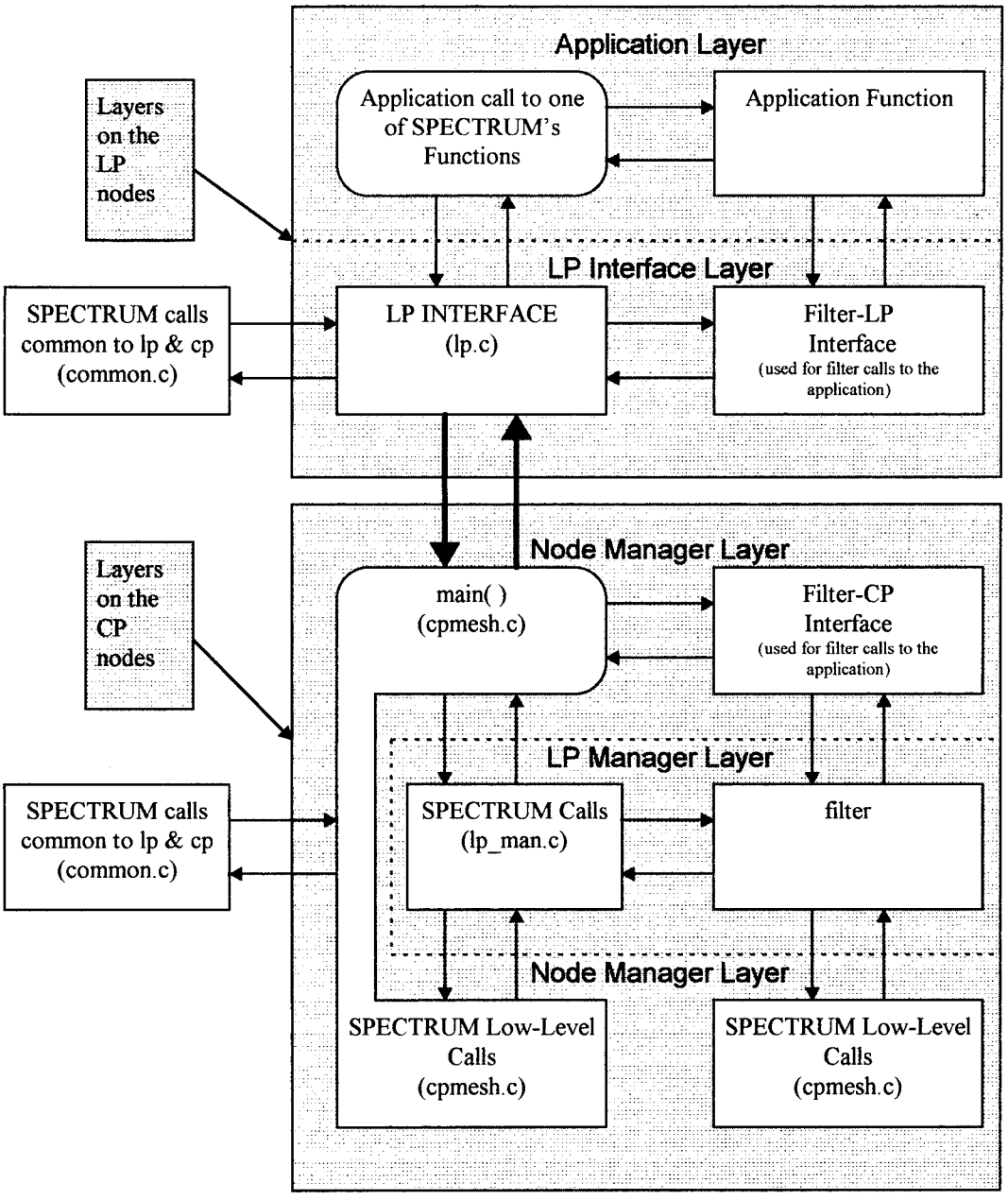

Figure 17: CPSPECTRUM's Layers

Two of CPSPECTRUM's layers run on the coprocessor: the Node Manager Layer and the LP Manager Layer. These layers allow the CP to service requests from the LP. Both the LP node and the CP node have main( ) functions. The LP's main( ) function is in the Application Layer and the CP's main( ) function is in the Node Manager Layer. The main( ) functions are indicated by rounded boxes in Figure 17.

4.3.3 LP Interface Layer

The LP interface layer is designed to replace SPECTRUM's functions on the LP with small functions that send requests to the coprocessor. To maintain compatibility with existing simulations, the functions supported by the LP interface layer have the same names and parameters as the original SPECTRUM. They also return the same values as SPECTRUM's functions.

The Filter-LP Interface is used to maintain compatibility with existing filters. When the filters were moved to nodes that were not running the application, filter calls to the application layer posed a difficult problem . Since the filters, to maintain backward compatibility, could not be modified, CPSPECTRUM had to provide some mechanism by which the coprocessor could emulate the application's response to a function. The Filter-LP Interface is part of that mechanism. Whenever the application makes a call to one of SPECTRUM's functions, the LP Interface calls a function in the Filter-LP Interface. The Filter-LP Interface function names are predefined as shown in the following table:

SPECTRUM Call	Filter Used	Interface Functions
lp_init( )	INIT_FILTER	init_filter_interface_lp( )
		init_filter_interface_cp( )
lp_post_event( )	POST_FILTER	post_filter_interface_lp( )
		post_filter_interface_cp( )
lp_post_message( )	MESS_FILTER	mess_filter_interface_lp( )
		mess_filter_interface_cp( )
lp_advance_time( )	TIME_FILTER	time_filter_interface_lp( )
		time_filter_interface_cp( )
lp_get_event( )	GET_FILTER	get_filter_interface_lp( )
		get_filter_interface_cp( )
lp_terminate( )	TERM_FILTER	term_filter_interface_lp( )
		term_filter_interface_cp( )

Table 5: Filter Interface Functions

In Table 5, two filter interface functions are shown for each filter. The functions whose names end with "_lp" are run on the LP node, and the functions whose name end with "_cp" are run on the CP node. Whenever a Filter-LP Interface function is called, it calls any application-level functions or examines an application-level global variable and places any values that the filter will need in a message. If the filter does not need any information, it returns a null pointer. This message is then passed to the CP node, along with any of SPECTRUM's variables that the LP manager's function will need, telling the CP to perform one of SPECTRUM's services. The LP will therefore be able to pass the filter any information it needs, but the filter will not be able to pass any information back to the LP, except what it embeds in an event returned by lp_get_event( ). The filter's inability to send information directly to the LP is not a problem since the filter sends the information back in an event.

The LP Interface layer also has a modified copy of node_level_init( ) which is used to initialize the LP's data structures and start the simulation. In addition to starting the simulation,

node_level_init( ) is used to load all of the coprocessors onto their nodes and sets the control node. Another similar copy of node_level_init( ) initializes the CP. Both of these versions of node_level_init( ) map the LPs and CPs to nodes. To avoid having two copies of the same source code, these mapping routines were moved to a function called node_map( ) and this function was placed in common.c, which will be described later.

4.3.4 CP Layers

The CP has two layers: the node manager layer and the LP manager layer. The node manager layer contains SPECTRUM's normal functions with several extensions. A main( ) function was added to the node manager so that it could start the coprocessor by performing the same functions as lp_level_init( ) and then calling node_level_init( ). The function, node_level_init( ), was modified so that it called lp_init( ) to run the INIT_FILTER, and started a loop where the CP received messages. The node manager in the coprocessor can receive both messages from other CPs, such as events and synchronization messages, and messages from the LP for which it works.

The coprocessor allows the filter to access data from the application layer through the Filter-CP Interface functions. There are two types of Filter-CP Interface functions. The first type of Filter-CP Interface functions are defined in Table 5 and are used to retrieve information from the LP's requests. These functions take a buffer as an input and assign the information in the buffer to the appropriate variables. If any of these variables are copies of application-level global variables used by the filter, then the Filter-CP Interface must declare them. The second type of Filter-CP Interface function is a duplication of a function in the application level. It has the same parameters and returns the same type variable as the function in the application level, but its body is different. The body of this second type of Filter-CP Interface function simply returns a variable

which was set by the first type of Filter-CP Interface function. Although these functions add to the coprocessor's complexity, they allow filters and applications to be run on the coprocessor without modification. The only changes to the application's code are minor changes in its make file. For every new filter that is going to be used with the CP, a Filter-LP Interface and a Filter-CP Interface is also needed. Fortunately, these files are fairly simple and the new files can be created by altering other filter's files.

4.3.5 Functions Common to LPs and CPs

Several of SPECTRUM's functions are used by both the LP and the coprocessor. These functions are shown in Figure 17 as two blocks that are outside of CPSPECTRUM's layers and are defined in the file, common.c. Most of the common functions are utility functions from SPECTRUM's LP manager layer. Two of the functions in common.c pass messages between the LPs and their CPs. Another function, called node_map( ) maps the LPs and CPs to the Paragon's nodes. Since both the LPs and CPs know which node is acting as the control node, set_control_node( ) is included in common.c so that they can both set the control node.

The utility functions included in common.c are functions that were previously defined in the LP manager layer. Since the LP Manager layer was moved to the coprocessor, and since the LP might still use these functions, they were block-copied from lp_man.c to common.c. These utility functions are: read_lp_info( ), display_lp_info( ), open_file( ), errlog( ), display_event( ), and log( ). All of these functions are exactly the same as their SPECTRUM counterparts, with the exception of log( ). It has been modified to write to separate files for the LPs and CPs.

Two of the functions in common.c, send_message( ) and recv_message( ), are used to pass messages between the LPs and CPs. Given a message type, a SPECTRUM buffer, a user buffer, and the lengths of the buffers, send_message( ) will send a message between an LP and a

CP. A global variable tells send_message( ) whether it is being run on an LP or a CP, and it automatically sends the message to the appropriate node. The receiving node calls recv_message( ), passing to it the type of message to be received. This function returns a message structure to the receiving node. The message structure contains a SPECTRUM buffer, a user buffer and the buffer lengths. The SPECTRUM buffer is intended to allow CPSPECTRUM to pass information from the LP Manager and Node Manager between the LPs and CPs. The filter interface functions employ the user buffer to pass application-level data to the LP Manager and Node Manager layers. These message passing functions simplify the coprocessor's software by providing a consistent, but versatile, interface between the LPs and CPs.

The functions node_map( ) and set_control_node( ) initialize both the LPs and the CPs. Because both the CPs and LPs can access the lp_info_table, they both have to run the distributed mapping algorithm that assigns the LPs and CPs to nodes. The mapping algorithm that SPECTRUM used in node_level_init( ) was copied into node_map( ) and placed in common.c so that both types of nodes could run it. Since both nodes have access to the lp_info_table, they also need to know which of the nodes is acting as the control node. The control node's CP number is set in the header file, mesh.h, as a constant and set_control_node( ) uses this constant to set the control node's number and ptype.

4.3.6 Types of Messages Passed Between LP and CP

After determining which functions were needed in the LP Interface Level, it was necessary to define messages with which the LP Interface Layer could make calls to CPSPECTRUM on the CP. It was also necessary to determine which of CPSPECTRUM's functions needed to pass information back to the LP. The messages passed to and from the CP are shown in Table 6. Each of the LP manager's functions, except lp_level_init( ), correspond to a message type used to

request that function from the CP. All of these messages are passed using send_message( ) and recv_message( ).

Function	Message Types	Request Contents	Reply Contents	Comments
node_level_init( )	Request: START_LP			This type of message is sent from the CP to the LP at the beginning of the simulation.
lp_level_init( )	none			Performed on both LPs and CPs
lp_init( )	Request: INIT_REQ	LP #		Tells CP to initialize filters
lp_post_event( )	Request: POST_EVENT_REQ Reply: POST_EVENT_RPL	Event		
lp_get_event( )	Request: GET_EVENT_REQ Reply: GET_EVENT_RPL		Event or null pointer	Null pointer is used in simulations with both application and SPECTRUM next event queues. Null pointer is returned when it is safe for the application to process the next event on its queue.
lp_advance_time( )	Request: ADVANCE_TIME_REQ Reply: ADVANCE_TIME_RPL	New Time	my_clock, ok_to_advance, time_to_advance	This function does not return the values sent in the reply. Instead, these values are updated in the appropriate global variable before the function returns.
lp_terminate( )	Request: TERMINATE_REQ	Event		Assumed to be the last message sent to the CP from the LP.

Table 6: Messages passed between the CP and the LP

As Table 6 suggests, only two of CPSPECTRUM's functions send reply messages to the LP. Reply messages lower the amount of parallelism that the CP can exploit because they force the LP to wait until the CP replies. More parallelism can be exploited if the LP can send a request to the CP and then continue to process events while the CP handles the request. Unfortunately, there are several circumstances where the LP needs to wait for a reply, such as when the LP requests the next event from the CP using lp_get_event( ). The LP must wait for the event; it would not have requested the event unless it needed it. The only other function that sends a reply to the LP is lp_advance_time( ) which sends the updated value of my_clock and several other

values to the LP. Ideally, the LP would know my_clock's value from its next event, but some simulations, such as the carwash, rely on my_clock to determine when to terminate.

4.3.7 Global Variable Concerns

SPECTRUM used global variables extensively. These global variables reduced the code's clarity and complicated the coprocessor's design. Many of these variables were used by several files, across many of SPECTRUM's layers. An example of such a variable is my_clock which SPECTRUM uses to store the LPs' local simulation times. In the carwash simulations, the application layer used my_clock to determine when to end the simulation, the filters used my_clock to maintain causality, and the LP Manager layer updated the value of my_clock. If a global variable was accessed by both the application and SPECTRUM, moving that variable to the coprocessor would require some mechanism to insure that the variable held the same value on both the LP and the CP. Fortunately, my_clock was the only variable both the LP and the CP could access. Either, not both, the LP or the CP used all of the other global variables.

To make sure that my_clock was consistent between the LP and the CP, this research had to examine when the variable was assigned values and when the variable was read. The only function that wrote a value to my_clock was lp_advance_time( ). This function wrote to the variable when it updated the simulation clock. In the carwash, my_clock was read after lp_advance_time( ) was called. Since lp_advance_time( ) would be moved to the coprocessor, it would have to send a message to the LP after the clock advanced, telling the LP my_clock's new value. When the LP called lp_advance_time( ), it would have to send an ADVANCE_TIME_REQ to the CP and wait for an ADVANCE_TIME_RPL from the CP.

Some of the SPECTRUM's global variables were declared as static in the header file, mesh.h. These variables could be a source of significant problems, since the static declaration in

the header file would make them local to each file that used them. For example, if function A( ) set variable X in file I.c and function B( ), which was in file J.c, tried to read the variable, B( ) would read a different value for X. Since functions in several files used the global variables in mesh.h, they are no longer declared as static.

4.3.8 Initialization and Termination Algorithms

Moving SPECTRUM to a coprocessor also required changes to its initialization and termination algorithm. The initialization algorithm needed to be changed to include assigning LPs to CPs, loading the CPs, and initializing them. The termination algorithm had to be changed so that the LP could shut down while the CP waited for the other LPs to finish.

A CPSPECTRUM-based application consists of two executable files and at least one data file. One of the executable files is run on the LP nodes, and the other executable file is run on the CP nodes. The data file contains information about the simulation's communication arcs. The name of the LP's executable file is based on the simulation's name. The name of the CP's executable file is defined in the Filter-LP Interface by the string, cp_program, and is typically named by appending "_cp" to the filter's name. For example, the carwash can be run with several filter executables called: nullwash_cp, delwash_cp, safewash_cp and sradwash_cp.

To simplify CPSPECTRUM's use, the LPs load each CPs' executable onto the appropriate nodes. This prevents the user from having to explicitly load all of the CPs from the command line. To start a simulation, the user types the name of the LP's executable followed by any command line arguments and the number nodes needed (using the -sz switch). This command will load the LPs on all of the nodes allocated to the simulation. The LPs will assign functions and arguments to an array of function pointers and an array of arguments. Like SPECTRUM, the LPs will call lp_level_init( ) and that function will call read_lp_info( ) and node_level_init( ).

When node_level_init( ) is called, the LP will allocate memory for all of CPSPECTRUM's mapping variables and will map the LPs and CPs to nodes using node_map( ). Once all of the nodes have been mapped, node_level_init( ) will do one of two things, depending on whether or not the node is supposed to be a coprocessor. On nodes that are supposed to be LPs, node_level_init( ) will start executing the LP's code or, on nodes that are supposed to be CPs, it will send a message to one of the LPs and exit. It is important to note that the LP's executable is loaded on all of the nodes and exits, before the simulation begins, on nodes that are supposed to be coprocessors. One of the LP processes, which has a node number of 0 and a ptype of 0, is used to load all of the CPs on the appropriate nodes. This node waits for messages from the LPs that exit on the CP nodes to insure that it does not load the coprocessors over LPs while they are still running. Node 0, ptype 0 was selected to load the CPs because it is always present in any Paragon application. If a program on the Paragon consists of only one node, it will still have a node 0, ptype 0. With the way in which CPs are mapped to nodes, node 0, ptype 0 will always be a LP. Once this LP has loaded all of the coprocessors, all of the LPs wait for a message of type START_LP from their coprocessor. When this message is received, the LPs call the application's LP functions and the simulation begins.

When a CP is loaded, it calls read_lp_info( ) and then calls the CP's version of node_level_init( ). Like the node_level_init( ) used on the LPs, this version of the function initializes the mapping variables using node_map( ). It differs from the LP version of node_level_init( ) in that it does not call the application functions. Instead, it sends a START_LP message to the LP for which it works and enters a while loop. This while loop calls node_receive_pending_messages( ) repeatedly until the simulation is completed. Since node_receive_pending_messages( ) can receive messages from both the coprocessor's LP and from other CPs, this loop handles all of the CP's tasks while the simulation runs.

With CPSPECTRUM, the simulation terminates at the same time it would when using the normal version of SPECTRUM. When the LP's local simulation time is greater than or equal to the stop time, the LP calls lp_terminate( ). When it calls lp_terminate( ), the LP sends a message, with an event embedded in it, telling the CP that it is terminating. Once it has sent this message to the CP, the LP exits. The coprocessor calls lp_terminate( ) when it receives the message, and the simulation terminates the same way it would with SPECTRUM, calling node_terminate( ) and shut_down( ). The coprocessor does not exit until directed by the control node. It waits for messages and continuously empties its message buffer until it is told to exit and in doing so, allows the simulation to terminate cleanly.

4.4 Implementation

CPSPECTRUM's implementation involved modifying SPECTRUM Node Manager Functions, grouping the functions common to both the LP and CP in a separate file, adding several new mapping variables, and modifying the simulation's make files. Some of the more significant modifications are described in this section of the report and several parts of the source code are explained.

4.4.1 Loading Coprocessors

Several techniques for loading the coprocessors were examined before a suitable one was found. Originally, each LP was designed to load its own coprocessor using the exec( ) function. This approach did not work because each LP used exec( ) to load the same coprocessor, but on different nodes. Because it supports distributed process groups, the Paragon's operating system realized that multiple copies of the coprocessor were being placed in the same process group and killed the simulation. Because the coprocessors were loaded separately, by the LPs for which they

would work, the Paragon's operating system would not allow them to talk to each other, even if it allowed them to be loaded using exec( ). The operating system terminated the simulation because all of the coprocessors needed to be loaded using a single command that would place the coprocessors on the correct nodes.

To fix the problems with loading the coprocessor, nx_load( ) replaced the exec( ) call, which is one of the calls in the Paragon's NX message passing library. The nx_load( ) function, given the node numbers, the number of nodes, the desired ptype, and the executables' name, loads the executable on the specified nodes. It also uses an array of type `pid_t`, to return the coprocessors' process identifiers. Because nx_load( ) loads all of the coprocessors, node number 0, ptype 0, loaded the CPs. The code used to load the coprocessors is shown in Figure 18. This routine waits for messages of type READY_FOR_NXLOAD from all of the LPs that exit on the CP-nodes, and creates an array of node numbers, called cpnodes, on which the CPs will be loaded. It then loads the CPs and checks to see if they loaded correctly. If the Paragon was unable to load all of the CPs, this routine terminates the simulation and displays an error message.

```
#ifndef IPD
    /* if Interactive Parallel Debugger is not going to be used */

    /* nx_load() the coprocessor program on the cp nodes */
    if (mynode() == 0 && myptype() == 0)
       {
        for (i=0; i < NUM_PROCS; i++)
          {
           cpnodes[i] = cp_locate_table[i].CP_Node_Index;
           /* make sure that lp program on cp nodes has exited */
           crecv(READY_FOR_NXLOAD,  &msg, 0);
          }

        printf("\nNode (%d) will be executing cp program: %s.\n", mynode(),
            cp_program);
        i = nx_load(cpnodes, NUM_PROCS, 0, pids, cp_program);
        if (i == -1)
          {
           printf("ERROR -- errno = %d\n", errno);
          }
        else if (i != NUM_PROCS)
          {
           printf("ERROR -- was only able to load %d cps -- ABORTING\n", i);
           errlog("ERROR -- was not able to load all of the cps");
           kill(0, SIGKILL); /* This call kills everything in process group */
          } /* end if */
       } /* end if mynode() ==0 && myptype() == 0 */

#endif
```

Figure 18: Loading the Coprocessors

The source code in Figure 18 uses a flag, called IPD, to determine whether the code that loads the coprocessors should be compiled. This flag was added so that the Paragon's Interactive Parallel Debugger (IPD) could be used with the coprocessor. IPD was unable to debug the program when it called nx_load( ) because it cannot debug processes in a dynamic process group. IPD requires a static process group and requires that the coprocessors be loaded manually, before the simulation is started. The process of loading the coprocessors manually is described in Appendix A.

4.4.2 Mapping Variables

Moving SPECTRUM's functions to the coprocessor created the need for additional mapping variables that allowed CPSPECTRUM to keep track of its nodes. These variables tell

the LPs which CP they are supposed to use and tell the CPs for which LP they work. Table 7 summarizes the mapping variables which are set using the function node_map( ).

Name	Type	Purpose
NUM_PROCS	int	The number of LPs. Since the number of CPs is equal to the number of LPs it is also the number of CPs.
lp_mesh_table[NUM_PROCS]	LP_TO_MESH	Indexed by LP number. Points from LP to node, ptype and to CP's node, ptype.
cp_mesh_table[NUM_PROCS]	CP_TO_LP	Indexed by LP number. Points from CP to LP's node, ptype.
cp_locate_table[NUM_PROCS]	NUM_TO_CP	Indexed by CP number. Points from CP number to CP's node, ptype
lp_num_table[number of nodes]	int	Indexed by node number. Holds the number of LPs per node. For a CP's node number, this variable is set to -1
my_lpid	int	The LP's node number. On a CP, it is the LP number for which the CP works.
my_cpid	int	The CP's node number. On a LP, it is the CP number that the LP uses.
I_AM_LP	int	Set to 1 on an LP. Set to 0 on a CP.

Table 7: CPSPECTRUM's Mapping Variables

4.4.3 CP Message Passing

As mentioned in the low-level design, the LPs and CPs pass messages using the functions send_message( ) and recv_message( ). The coprocessor messages passed using these functions contain two parts: a SPECTRUM buffer and a user buffer. The format of the coprocessor messages is shown in Figure 19.

The coprocessor message structure contains two pointers that point to the SPECTRUM buffer and the user buffer. It also contains the buffers lengths. When send_message( ) is used to send a message between a LP and a CP, it must copy the entire message into a contiguous memory block so that the Paragon's operating system can send the message. Since the Paragon's csend( ) call, which is used to send a message, takes a pointer to an object and the object's size as input parameters, it does not know the contents of the messages and cannot follow the pointers in the coprocessor message to the buffers. When a message is placed into contiguous memory, it is

said to be "packed". The coprocessor messages are packed in the following order: the message structure, the SPECTRUM buffer, and the user buffer.

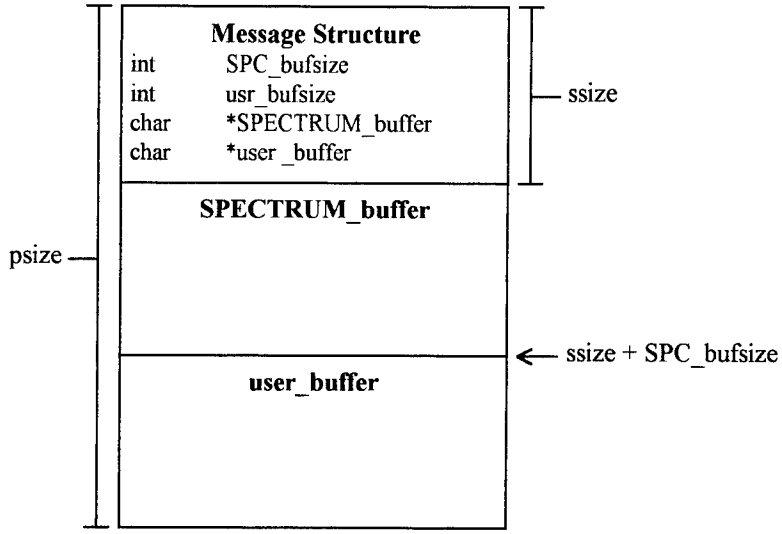

Figure 19: Coprocessor Message Format

When the recv_message( ) function is used to receive a message, the pointers in the message's structure are incorrect since the message has been packed. To unpack the message, recv_message( ) allocates enough memory for a message structure and it then copies the information from the packed message's structure to the unpacked message's structure. Using the buffer lengths from the packed message, recv_message( ) then allocates enough memory for the buffers and assigns the location of the buffers to the appropriate field of the unpacked message. The buffers in the packed message are then copied to the unpacked message's buffer. The SPECTRUM buffer begins immediately after the message structure at *ssize* in Figure 19, where *ssize* is the structure's size. The user buffer begins after the SPECTRUM buffer, at *ssize* + *SPC_bufsize*, where *SPC_bufsize* is the size of the message's SPECTRUM buffer.

4.4.4 Preventing Recursive LP Requests

CPSPECTRUM was designed to detect many types of error conditions. One such error condition would occur if a LP request was received while another LP request was being processed. When the LP and CP are running normally, this type of error is impossible, but it could occur if some type of error condition existed that caused to LPs to send requests to the same CP.

During the simulation, the CPs can only receive messages when they are running the function, node_receive_pending_messages( ). When a CP receives a request from the LP for which they work, node_receive_pending_messages( ) calls the LP manager function appropriate for the request and the LP either waits for a reply or continues running the simulation. When a CP receives an event from another CP, node_receive_pending_messages( ) calls lp_post_message( ) to post the event in SPECTRUM's next event queue. Throughout the CP's execution, only one thread of control is used, and that thread switches between servicing LP requests and receiving events from other coprocessors as needed. Most of the LP manager functions do not call node_receive_pending_messages( ), but if they do then the CP can receive another request from an LP that will cause them to process the second request before they finish processing the first request. Such a recursive call to node_receive_pending_messages( ) could cause causality errors or deadlock. To prevent such a recursive call, all of the LP manager's functions were examined. Only lp_get_event( ) called node_receive_pending_messages( ) and might cause recursion. A more detailed examination of both the LP's and the CP's algorithm revealed that lp_get_event( ), as it was currently implemented, would not cause recursion. Recursion was avoided because the LP, after requesting that the CP execute lp_get_event( ), had to wait for the CP to return an event before it continued running the simulation. Recursion was prevented in lp_get_event( ) because the LP would not send another request to the coprocessor until the CP completed its current task

and returned an event to the LP. Therefore, the only way recursive calls could be made to the LP manager would be if more than one LP sent requests to a single CP or if an LP's code executed incorrectly.

Even though recursive calls to the LP manager would not occur when the program ran correctly, CPSPECTRUM was modified to detect such a recursive call and terminate the simulation when one was detected. This extra error checking was simple to implement and could prevent many problems. CPSPECTRUM prevents recursive calls to the LP manager by setting a flag, called performing_cp_function, every time the CP processes a request from the LP and clearing the flag when the CP is finished processing the request. If the CP receives a recursive request for one of the LP manager's functions, the CP will see that the flag has already been set and terminate the simulation.

The error checking that prevents recursive calls to the LP manager's functions is not necessary for the other simulation functions. In fact, checking all of the received messages for recursive calls to node_receive_pending_messages( ) would prevent the simulation from running. When a CP receives a request for lp_get_event( ), this LP manager function will recursively call node_receive_pending_messages( ). This recursive call can be seen in Figure 20, which is a simplified flow of control diagram for the coprocessor. The ellipses in this diagram represent places where unnecessary detail was omitted. When node_receive_pending_messages( ) receives an event, it will return to lp_get_event( ) which will return an event to node_receive_pending_messages( ) that will be sent back to the LP. Since the recursive call to node_receive_pending_messages( ) returns before it can be called again, this recursion will only occur two levels deep in the coprocessor.

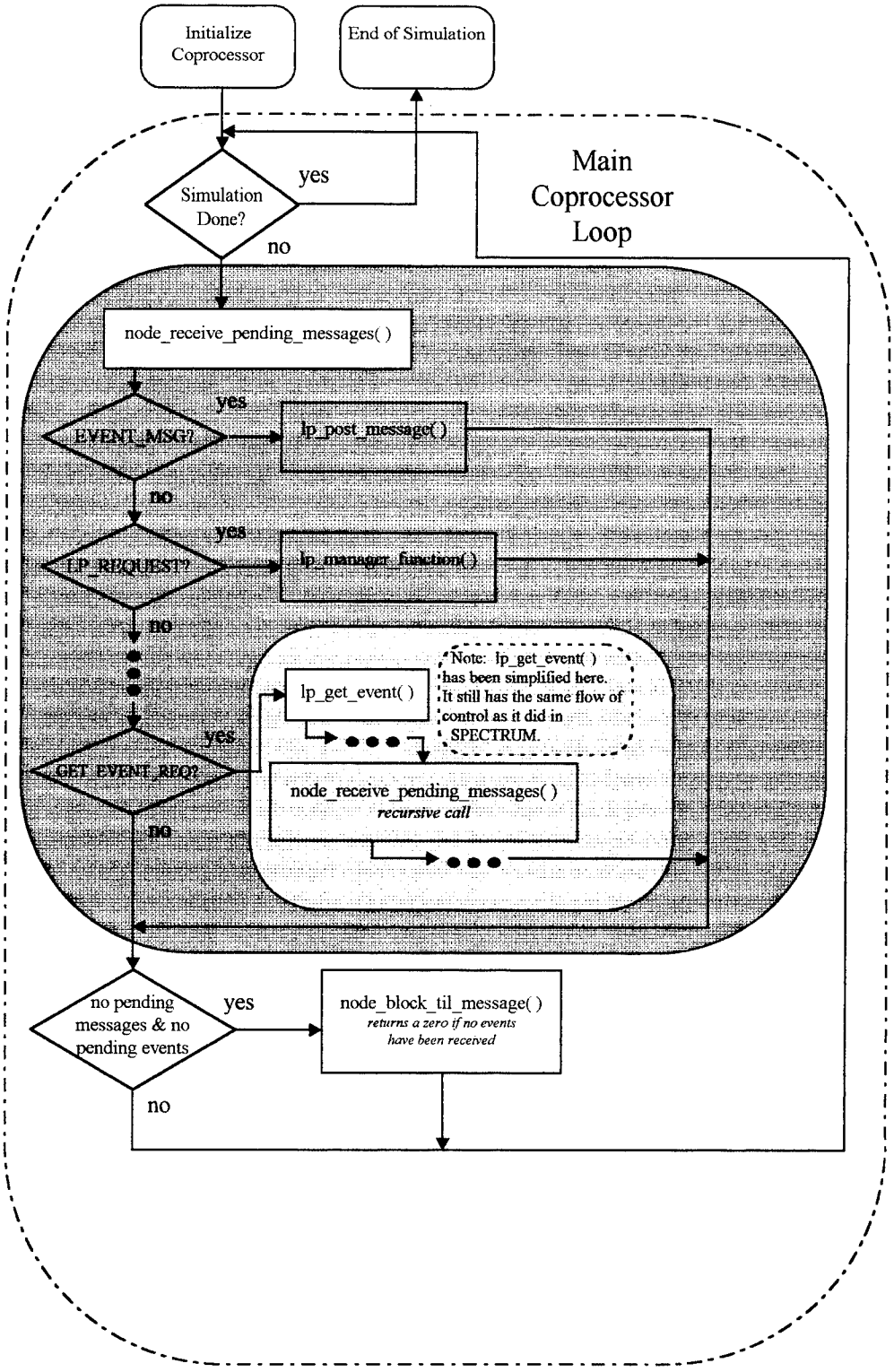

Figure 20: Simplified CP Flow of Control

4.4.5 Example Interface Functions

Although five filter interfaces had to be written for carwash and VSIM to run using CPSPECTRUM, only VSIM's filter interface had to pass data from the LP to the CP. VSIM's filter interface had to be written to allow its filter to call the function get_low_time( ) in its application layer. The function get_low_time( ) is called by VSIM's GET_FILTER and is supported by the function get_filter_interface_lp( ), which is shown in Figure 21. This Filter-LP Interface function is run on the LP whenever the LP calls lp_get_event( ). It calls get_low_time( ) and places the value returned by that function in a message. When the LP sends the request, it includes the message sent by the Filter-LP Interface Function.

```
/*****************************************************************************
* Function: get_filter_interface_lp
* Purpose: This interface function places information in the user buffer of a
*          message which is sent to the coprocessor.  The information in the
*          user buffer is used by the GET_FILTER on the coprocessor.  The
*          information placed in the message is dependent on the application
*          being run.
*
* Parameters: returns a struct message
* Comments: This function is intended to be run on the lp node.
* History: created 5 September 94 Walton
*
*****************************************************************************/
struct message get_filter_interface_lp()
{
struct message    the_message;
unsigned int      the_time;

   the_message.user_buffer = (char *)malloc(20);
   if (the_message.user_buffer == NULL)
     {
      printf("Out of memory on CP node(%d) ptype(%d)\n",
              mynode(), myptype());
      errlog("CP Out of memory in get_filter_interface()");
      node_abort();
     }

   the_time = get_low_time();
   sprintf(the_message.user_buffer, "%u", the_time);
   the_message.usr_bufsize = strlen(the_message.user_buffer) + 1;

   return the_message;
} /* end get_filter_interface_lp */
```

Figure 21: Example Filter-LP Interface Function

When the coprocessor receives a request from the LP for which it works, it calls the appropriate Filter-CP Interface function. Figure 22 shows VSIM's Filter-CP Interface function for GET_EVENT_REQ messages. This function uses sscanf( ) to take a value out of a string passed to it from the Filter-LP Interface and places that value in a global variable.

```
/*****************************************************************************
* Function: get_filter_interface_cp
* Purpose: This interface function takes the information in the buffer which
*          is passed to it and assigns it to the appropriate variable.  The
*          filter, which is running on the cp, can the use the variable with-
*          out any modifications to the filter.  This function takes the
*          information in the buffer and assigns it to variables used by the
*          GET_FILTER.  The variables written to by this routine depend on
*          the application being run.
*
* Parameters: the size of the user_buffer and the user_buffer
* Comments:
* History: created 5 September 94 Walton
*
*****************************************************************************/
void get_filter_interface_cp(bufsize, the_buffer)
int  bufsize;
char *the_buffer;
{
   sscanf(the_buffer, "%u", &the_low_time);

   return;
} /* end get_filter_interface_cp */
```

Figure 22: Example Filter-CP Interface Function

Both the Filter-CP Interface function and a stub function use the global variable, the_low_time, which returns its value to the GET_FILTER when it calls get_low_time( ). Since the filter is the same as the one used by SPECTRUM, it calls get_low_time( ) as if it could make the call directly to the application. The stub function must therefore have the same name and parameters, and must return the same type of value as its application-level counterpart. The stub function for get_low_time( ) is shown in Figure 23. All this function does is return the global variable, the_low_time, but in doing so, allows the filter to make an indirect call to the application layer.

```
/*****************************************************************************
* Function: get_low_time()
* Purpose: This 'stub' function duplicates a function call in SPECTRUM's
*          application level so that the filter is allowed to call it.
*
* Parameters: returns an unsigned integer which is equal to the lowest time
*             in vsim's next event queue when the application made a call to
*             lp_get_event()
*
* Comments: uses data passed by get_filter_interface_lp and
*           get_filter_interface_cp
*
* History: created 5 September 94 Walton
*
*****************************************************************************/
UINT32 get_low_time()
{
return the_low_time;
} /* end get_low_time */
```

Figure 23: Example Filter-CP Stub Function

4.5 Summary

CPSPECTRUM was developed using a three-stage approach. First, a high-level analysis was performed on SPECTRUM's functions, variables and layers. Next, a low-level design specified CPSPECTRUM's algorithms, layers and structures, and then these specifications were implemented. Several new data structures were needed to map LPs and CPs to one another and the Paragon's nodes, and several new functions were needed. Most of the changes to the source code were made in SPECTRUM's node level, and several new files were necessary to simulate SPECTRUM's presence on the LP and to allow the filters to make calls to the application layer. Once CPSPECTRUM was implemented, the simulations' make files were altered and tested. The next chapter describes the methodology used to examine the coprocessor's effect on the carwash and VSIM simulations. The coprocessor was tested using the carwash simulation, and several VSIM simulations, including the Wallace tree multiplier, and the associative memory simulations.

V. Test Methodology and Results

5.1 Introduction

This chapter describes how CPSPECTRUM's performance was measured and analyzed. It also includes the performance measurements used to conduct the analysis. Many factors can effect CPSPECTRUM's performance. Message passing latencies delay messages passed between the LPs and the coprocessors. These latencies were measured to model their effect on the simulations. The simulation's granularity was varied using spinloops to measure its effect on the CPSPECTRUM's performance. The VHDL simulations also used several circuit partitioning schemes to determine their effect on SPECTRUM's performance. Since the VHDL simulations can be partitioned to run on any number of nodes, the number of LPs was also varied. Varying all of these factors and measuring their effect on the simulation's performance provided several useful insights.

5.2 Message Passing Latencies

One disadvantage of implementing the coprocessors in software was that the messages passed between the LP and its CP would suffer from much larger delays than if a hardware coprocessor was used. Since a hardware coprocessor could be connected to the GP board's expansion slot, it could be accessed at the board's bus speed. To accurately model CPSPECTRUM's communications delay, a program was modified to time message latencies as they were passed in a ring. By timing the message's traversal of the ring, this program calculated the Paragon's communication bandwidth and message passing latencies.

The ring program varied the size of its message to see if it affected the message passing times. Node 0 started the ring program by sending node 1 a message. Node 1 and all successive nodes did nothing except receive the message and immediately send it to the next node. When the ring program was run on *n* nodes, the *n*th node passed the message back to node 0, thus forming a ring. To increase the times' statistical significance, the program passed the message around the ring 100 times for each message size. The ring program was run with two, eight and sixteen nodes to determine how varying the number of nodes affected the time.

The total times for the ring program are graphed in Figure 24. This figure illustrates many facts about the Paragon's message passing performance. It shows that the message passing times increased proportionally as the size of the ring increased. It took twice as long to pass a message around eight nodes as it did to pass the same message around four nodes. This figure also demonstrates the effects of message packetization. The Paragon breaks messages into 2 kilobyte packets in which the first 292 bytes are reserved for the system[20]. When a message's size exceeds 1,756 bytes, the message is broken into two packets. Figure 24's times have a step-like shape because of the packetization. Clearly, the Paragon takes longer to format a message than it takes the hardware to pass it.

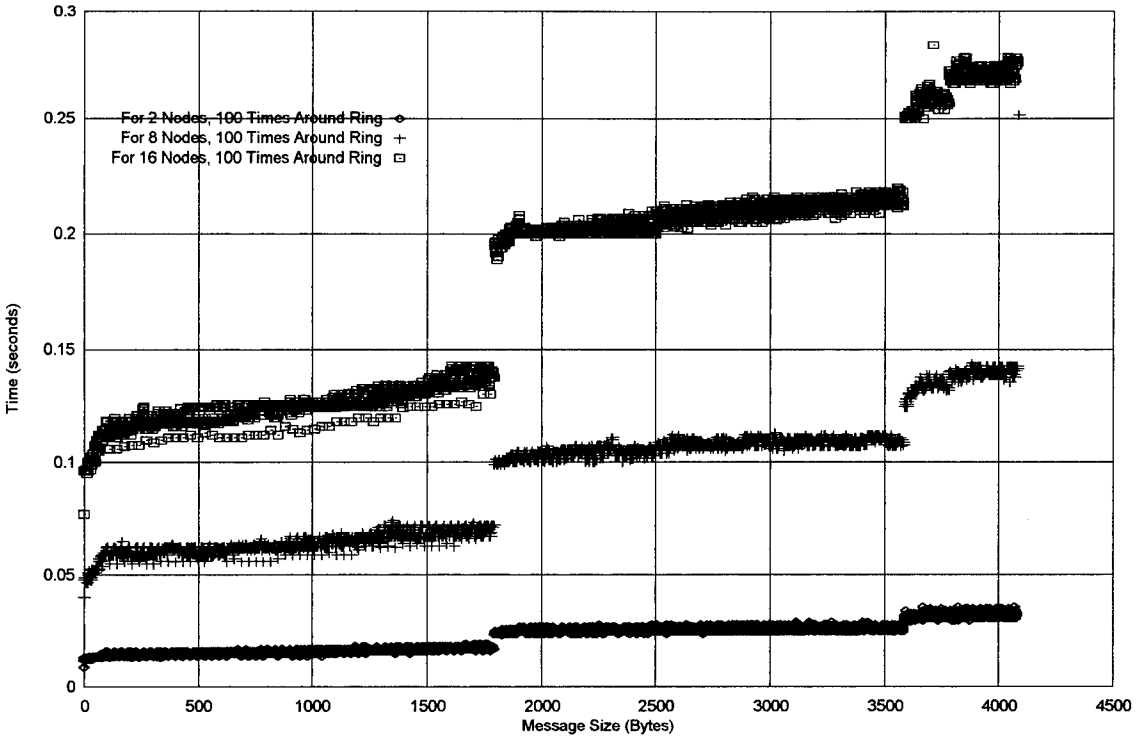

Figure 24: Total Message Times for Ring Program

The Paragon's message passing bandwidth is also affected by the message's size. The ring program measured how the message's size influenced the bandwidth and these results are shown in Figure 25. The ring program calculated the bandwidth by using the following formula:

$$bytes / second = \frac{msg\_size \times number\_of\_nodes \times number\_of\_times\_around\_ring}{ring\_time}$$

Equation 3: Bandwidth Calculation for Ring Program

Since the number of nodes and the number of times around the ring are factored into this formula, their effects are not shown in Figure 25. The bandwidth measurements for the ring program overlap and do not differ between the one, four and eight node runs. The graph of the Paragon's bandwidth reveals that its bandwidth increases rapidly as the message size increases. The large overhead in formatting messages causes this rapid increase in bandwidth. The breaks in the curve are due to packetization and illustrate the large overhead involved in breaking the message into

packets. The slope of the curve's segments decreases slightly as the message size increases because the larger messages tend to saturate the Paragon's communication's network. Eventually, as the message sizes increase, the Paragon's communications network will become saturated between 50 and 60 Mbytes per second [18]. Since the mesh routing components in the Paragon's hardware have a maximum bandwidth of 200 Mbytes per second, it is clear that the slower actual message passing times result from the way the operating system handles messages. Since most of SPECTRUM's messages are about 48 bytes long, the ring program was not used to examine where the Paragon's communication network saturated.

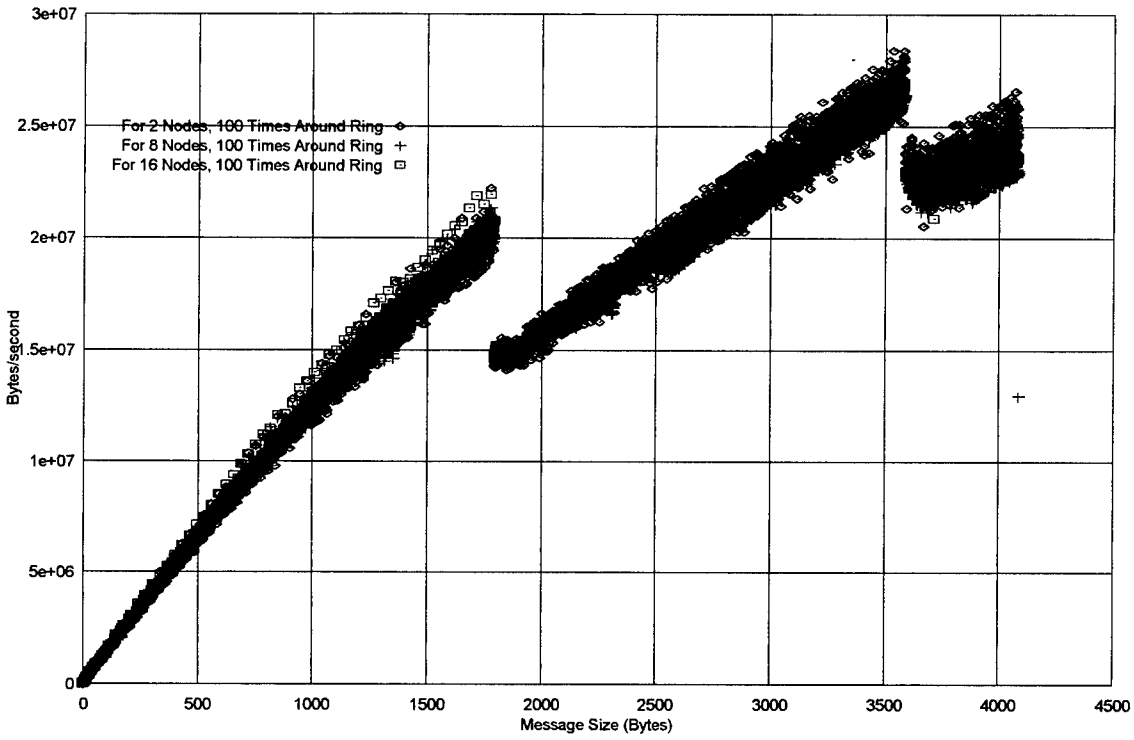

Figure 25: Paragon Communications Bandwidth

The ring program also recorded the average message passing times as the message size varied. These times are shown in Figure 26 and were calculated using the following formula:

$$average\_message\_time = \frac{ring\_time}{number\_of\_nodes \times number\_of\_times\_around\_ring}$$

Equation 4: Average Time to Pass Message in Ring Program

The message passing times in Figure 26, like the previous figures, have discontinuities as a result of packetization. The data graphed in Figure 26 indicates that most of SPECTRUM's messages take around 70 μseconds to pass between two nodes. Because most of the message passing latency results from the operating system and not the hardware, the message passing times between more than two nodes will also probably be 70 μseconds. Each mesh routing component the message passes through will add only 40 nseconds if the message passes straight through it and 180 nseconds if the message has to turn a corner in it [20]. Because the mesh routing component's delays are at least several orders of magnitude smaller than the message passing latencies, and because the Paragon's wormhole routing minimizes contention on the communication channels, these latencies do not have to be modeled to estimate the average delay for messages in a SPECTRUM simulation.

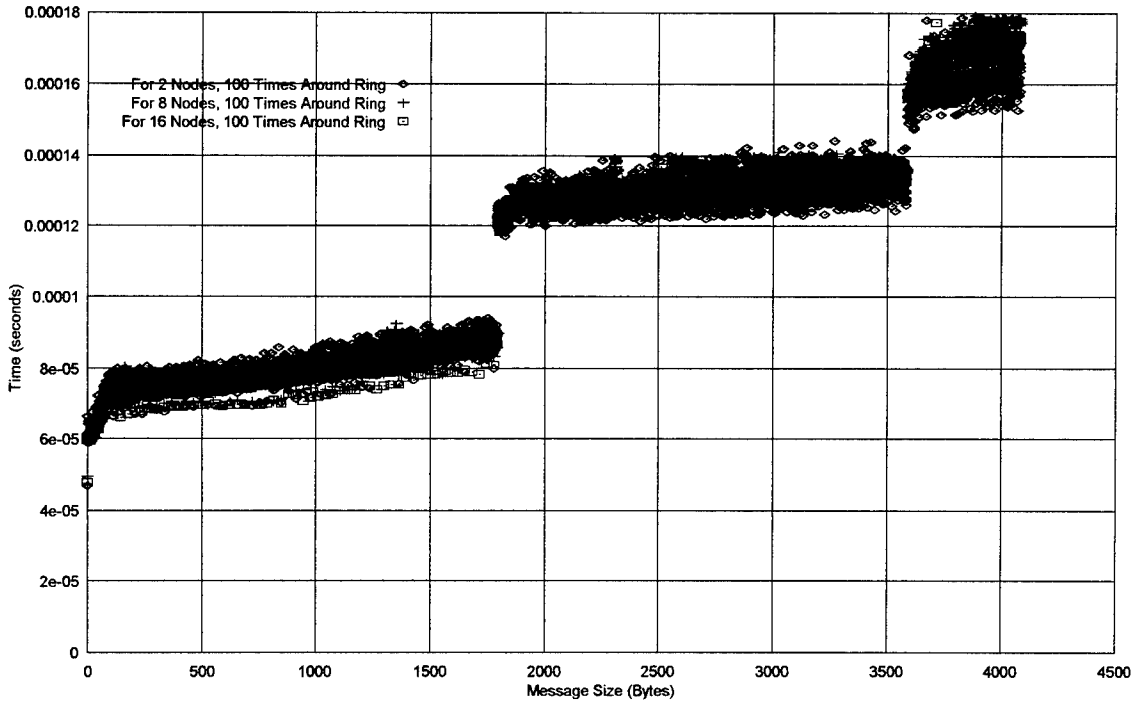

Figure 26: Paragon's Message Passing Latencies

5.3 Use of Spinloops

Spinloops revealed how the simulation's granularity affected the coprocessor's performance. The spinloop consisted of the code shown in Figure 27 and performed two floating point calculations for every loop. Floating point calculations were used because they require more time to execute. The first calculation performed in the spinloop was a multiply and the second was a divide. On the i860, division uses an iterative formula to take the inverse of denominator and then multiplies it with the numerator. This iterative inverse takes a relatively large amount of time and is ideal for adding a delay to a program. To prevent the number from overflowing or underflowing, the same number was used to divide as was previously used to multiply. Therefore the value of the variable, *u*, would not change after a spinloop completed. If *u* overflowed, an exception would be raised and the processor would have to perform extra work for that iteration of the spinloop. Since the variable is not changed, it did not overflow and the spinloop's execution time scaled linearly as the number of iterations were increased. If *u* was not used by any other part of the program, an optimizing compiler might eliminate the spinloop. To prevent the spinloop from being eliminated, *u* was assigned to another variable, called *v*, at the completion of the loop.

```
#ifdef SPIN
         for (i = 0; i < SPINLOOP; i++){
            u = u * 1.0000001;
            u = u / 1.0000001;
          }
         v = u; /* makes sure that compiler doesn't eliminate SPINLOOP */
#endif
```

Figure 27: Spinloop Code

The spinloops' effect on simulations depends on how much it delays the processor. The spinloop code in Figure 27 was timed, using the Reprogrammable Performance Monitor Clock, to determine how long the spinloop delayed the simulations. This program ran the spinloop 100,000 times and divided the final time by 100,000 to determine a single spinloop time. The program,

with a sample of 20 runs, measured the spinloop delay to be approximately 1.85 μseconds. Since the message passing latency was measured to be 70 μseconds, the spinloop would have to execute at least 38 times for each message to cancel the message passing latency's effect on the simulation's granularity. The average SPECTRUM simulation cycle consists of the following: lp_get_event( ), lp_advance_time( ), and an lp_post_event( ) for each new event to be scheduled. For a simple simulation like the carwash, it is relatively simple to figure out how many messages are passed to the CP per simulation cycle. The lp_get_event( ) call causes a request to be sent to the CP and reply to be received. The lp_advance_time( ) call also sends a request and receives a reply. The lp_post_event( ) only sends a request. If only one event is scheduled per simulation cycle, then five messages are used per cycle. To cancel out the message passing latency of five messages, the spinloop would have to be set to 38 spinloops/message × 5 messages = 190 spinloops.

For more complex simulations, such as VSIM simulations, the number of spinloops needed to cancel the message passing latencies would differ. For VSIM the number of spinloops needed to cancel the messages would be less than 190 because of VSIM's application-level NEQ. To determine the number of spinloops needed to cancel the message passing latencies, the number of events executed from the SPECTRUM next event queue for every event executed from VSIM's NEQ would have to be known. With an efficient partitioning strategy, most events would be executed from VSIM's NEQ and the number of spinloops needed to cancel the message passing latencies would be less than 190. Since CPSPECTRUM has to pack messages that it passes between the LPs and CPs, there is some overhead that is not counted in the spinloop timing measurements. This additional delay was neglected in this research because it was assumed to be relatively small compared to the message passing delays.

5.3.1 Adding Spinloops to Applications

For the carwash simulation, the spinloop was added to the application code in the file afit_wash.c. Since each LP had a separate function in the carwash, the spinloop had to be added to each of these functions. The spinloop was added using an #ifdef statement so that it would be compiled only if spinloops were requested. To include the spinloop code in the carwash, the flag SPIN had to be defined in the makefile for afitwash.c. Otherwise the spinloop would not be included and would not slow the simulation.

The spinloop was added to VSIM by modifying the file vsim.c. Like the carwash, the VSIM spinloop was added using an #ifdef statement so that spinloops would only be compiled if a flag, called BUSY, was set. VSIM's spinloop was added so that it delayed events from both VSIM's and SPECTRUM's next event queues.

5.3.2 Spinloop Input File

The original spinloop code used a constant, SPINLOOP, to define the number of times the loop iterated. The use of a constant was cumbersome because it required the program to be recompiled every time the size of the spinloop changed. To simplify altering the spinloop's size, the code was modified so that it read it from a file, spinloop.inp. This text file contained a single integer value that defined the number of spinloops. The node manager layer was modified so that the function, read_lp_info( ), read the spinloop input file and assigned its value to SPINLOOP.

There are several reasons why read_lp_info( ) was chosen to read the spinloop input file. Since read_lp_info( ) is executed before the nodes mapped to LP's, it reads the spinloop size before SPECTRUM calls fork( ) to create any addition LPs on a node. By reading the spinloop value this early in the simulation, it only has to be read by one process per node. When fork( ) is

called to create the additional LPs, they will already know the spinloop size. Since, in CPSPECTRUM, read_lp_info( ) is called by both the LPs and their coprocessors, the CPs will be able to record the spinloop size in their log files. This is advantageous because the simulation times are also written to the CP's log files. Storing both the simulation times and the spinloop size in the same file added to CPSPECTRUM's convenience. The code added to read_lp_info( ) is shown in Figure 28.

```
#ifdef SPIN
   fp = open_file("spinloop.inp",FREAD);
   fscanf(fp, "%d", &SPINLOOP);
   fclose(fp);
   if (SPINLOOP < 0) {
      errlog("Illegal SPINLOOP value in spinloop.inp");
      node_abort();
   }
#endif

#ifdef BUSY
   fp = open_file("spinloop.inp",FREAD);
   fscanf(fp, "%d", &SPINLOOP);
   fclose(fp);
   if (SPINLOOP < 0) {
      errlog("Illegal SPINLOOP value in spinloop.inp");
      node_abort();
   }
#endif
```

Figure 28: Code Used to Read Spinloop Input File

5.4 Granularity's Effect on Run Times

The extent of the parallel overhead affects a parallel algorithm's acceleration. Ideally, a parallel simulation's acceleration would increase linearly as the number of nodes used increased. With an ideal speedup, the simulation's time would be:

$$parallel\_time = \frac{sequential\_time}{number\_of\_processors}$$

Equation 5: Ideal Parallel Time

Unfortunately, most parallel simulations do not accelerate linearly with the addition of more processors. Instead, the additional processors cause the simulation to incur delays since the simulation has to keep the LPs synchronized. Including the parallel overhead, the simulation's time can be modeled using the following equation:

$$parallel\_time = \frac{sequential\_time}{number\_of\_processors} + parallel\_overhead$$

Equation 6: Realistic Parallel Time Model

In the above equation, the parallel overhead is the only unknown. The parallel and sequential times are easy to measure. Modeling the affects of the spinloops is not as simple as adding the spinloop delay to the above equation. If the spinloop was added to the parallel time equation, the resulting equation would look like the following:

$$parallel\_time = \frac{sequential\_time}{number\_of\_processors} + parallel\_overhead + spinloop\_delay$$

Equation 7: Incorrect Model for Spinloop Delay

This equation is not very useful because it contains two unknown values: parallel_overhead and spinloop_delay. Because the spinloops execute in parallel, the total delay resulting from the spinloops is hard to calculate. The spinloop delay for an individual event is easy to measure, and this delay can be multiplied by the number of events executed in the simulation to get a maximum total delay. This maximum total delay, however, assumes that all of the events are executed sequentially. If some of the events are executed in parallel, then the actual total delay as a result of the spinloops will be much less. The parallel overhead is unknown because it may vary as the simulation's granularity changes. The parallel overhead measurements from timing the simulation without spinloops will be different from the parallel overhead when spinloops are used.

Another, more feasible way to model the effects of spinloops is to measure their effect on the sequential simulation's time, and replace the sequential time in the equation with the sequential time when spinloops are used. The resulting equation is shown below:

$$parallel\_time = \frac{sequential\_time(\text{with spinloops})}{number\_of\_processors} + parallel\_overhead$$

Equation 8: Correct Model for Spinloop Times

This equation can be applied to the ideal parallel simulation by setting the parallel overhead to zero. Since the equation only has a single unknown, it can calculate the parallel overhead with spinloops. The parallel overhead, using this equation, is equal to:

$$parallel\_overhead = parallel\_time - \frac{sequential\_time(\text{with spinloops})}{number\_of\_processors}$$

Equation 9: Parallel Overhead

Measuring how the parallel overhead varies as granularity increases provides several fundamental insights into how the simulations run. The coprocessor's goal is to reduce the parallel overhead by relieving the LPs of the burdens of receiving messages and sending null messages. If receiving and sending messages take a large part of the LPs' times, then the coprocessor should accelerate the simulations. If the LPs spend the majority of their time blocking, waiting for a message from another LP, then the coprocessor will not provide much speedup. If the simulation's parallel overhead increases with the addition of spinloops, it indicates that the spinloops increase the amount of time that the LPs are blocked. Because the spinloops create a uniform delay for every event on every processor, it is likely that the time a blocked LP has to wait for an event to unblock it will scale linearly as the spinloops increase.

5.5 Use of the Graph Partitioning Tool

The Graph Partitioning Tool partitions the behaviors of a VHDL circuit to LPs. Kapp's thesis involved modifying the tool to use more efficient partitioning strategies and Joel Hurford took this work farther by further improving its partitioning strategies[25]. Several partitioning strategies were examined since they can have a significant impact on the simulations' performance. The first partitioning strategy examined was the random partition. This partitioning strategy randomly distributed VSIM's behaviors between the LPs. The random partition will almost always introduce feedback into the circuit and has a small lookahead. It was expected to perform very poorly.

Two of Hurford's improved partitioning strategies were also used in this research. The first strategy was a breadth-first search with no feedback loops. A feedback loop consists of a cyclic dependency and can result from the circuit's structure or the partitioning strategy. Feedback inherent in the circuit's structure cannot always be eliminated, but feedback introduced by the partitioning strategy can. Hurford eliminates feedback by grouping strongly connected components on a LP. Since feedback increases the amount of time the LPs spend blocking, the breadth-first search was expected to improve the simulation's performance.

The breadth-first search partitioning strategies, in addition to eliminating feedback, might pipeline the circuit. Pipelining could have one of several effects, depending on the nature of the circuit. If the circuit has feedback arcs from the end of the pipeline to the beginning, breadth first partitioning would negate any speedup gained from eliminating other, internal feedback loops. Since the first LP in the pipeline would have to wait for the LP at the end, the rest of the LPs in the pipeline would have to wait for the LP in front of them. In such a case, only one LP could

process events at a time. If no feedback path existed from the end to the front of the pipeline, then a large amount of parallelism would be exploited using breadth first search to partition the circuit.

The other partitioning strategy tested from Hurford's research was breadth-first search by source with no feedback. This partitioning strategy differs from the previous strategy only in the way the it attempts to distribute the sources among the LPs. By distributing sources evenly between the LPs, Hurford attempts to make sure that the LPs always have events in their next event queues. If the LPs in a simulation tend to block as a result of empty next event queues, this partitioning strategy will probably improve their performance. If the LPs tend to block while they wait for one of the LPs on their input arcs to catch up, having a source on the LP may improve performance because the source input arc is one less arc for which the LP has to wait.

5.6 Test Conditions

There are many ways in which the Paragon's operating environment can affect the simulations' run times. Since the Paragon is a multi-user system, any program it executes can influence the performance of other programs. Since the Paragon does not allow users to share a processor in its compute partition, the effect of programs competing for resources is not always obvious. When programs compete for resources on the Paragon, the most noticeable effect is that IO slows considerably. The most common form of conflict between two programs appears to occur when they write data through the boot node. Writing files or displaying text causes a program to perform inconsistently because it may have to wait for another program which is also sending information through the boot node.

To prevent boot node IO conflicts from skewing performance measurements, all output to the screen or to log files had to be either turned off or done outside of the timed portion of the

code. Since improving IO performance was beyond the scope of this thesis, all IO was done so that it did not affect the simulations' times. The LPs and coprocessors were loaded onto the nodes and initialized before the timing started and the final statistics were not written to log files or displayed until the simulation ended.

Since the log files were not written during the simulation, they did not affect the simulation's performance. They could still, however, affect other users if they were written through the boot node. To minimize the simulations' effect on other users, SPECTRUM executables were copied to disk drives, in High Performance Disk (HPD) space, local to the Paragon. HPD did not improve the simulations' run times, but it did improve the speed with which the simulation loaded.

One side effect of using CPSPECTRUM was that all of the simulations required twice as many nodes as they did with the normal version of SPECTRUM. For example, a 32-LP simulation required 64 nodes using CPSPECTRUM. Since the WPAFB Paragon only has 48 nodes in its open partition, some of the larger CPSPECTRUM simulations could not be run in it. Instead, the simulations were run in the Paragon's Network Queue Server (NQS). NQS has many queues with a variety of time constraints and node number constraints. Most of the simulations were fast enough to run in a fifteen minute queue and were small enough to run in less than 64 nodes. These queues were tested to ensure that they would not skew the simulation times by running several simulations in both the open partition and the queues. The queues did not appear to effect any of the simulation's times unless the queues ran two simulations in the same directory at the same time. When two such simulations were run simultaneously, the log files were interleaved and the simulation times were noticeably higher. To prevent two simulations from running simultaneously, the simulations were queued so that a program from another directory was queued between programs from the same directory.

The only other problem with using NQS to run large simulations involved the way the Paragon buffers files when they are written. SPECTRUM writes the simulation times to log files prior to exiting. When SPECTRUM was run in a NQS partition, these times would not get written because the partition would be removed by the operating system before the buffered write could be completed. To fix this problem, the sleep command was added to the scripts which ran simulations in NQS partitions. This sleep command is shown on the example script below:

```
cd /paragon1/waltonac/wallace/d20
cpwallace 2000 -sz 40
sleep 4
cpwallace 2000 -sz 40
sleep 4
```

Figure 29: NQS Script used to run Wallace Tree Multiplier

The first command in the script insures that the Paragon is in the correct directory. Then, the file cpwallace is executed using a run time of 2000 ns and 40 nodes. Since the coprocessor is in use, there are twice as many nodes as there are LPs. The file Wallace was renamed cpwallace so that the coprocessor version of the simulation could exist in the same directory as the normal version. Normally, the CPSPECTRUM simulation would have the same name as the normal version of the simulation and would exist in a separate directory. The coprocessor executable for VSIM, called vsim_cp, also had to be copied into the directories in HPD. Both the LP and the CP executables had to be recompiled every time the number of LPs changed. The number of LPs was varied by changing two lines of the file application.h, shown below:

```
/*************************************************************************
 * File:  application.h                                                  *
 *                                                                       *
 * Used for SPECTRUM interface to VSIM.  Has vspec.c globals and         *
 * prototypes.                                                           *
 *************************************************************************/
#define NUM_PROCS 8              /* number of logical processes */
#define INPUT_ARCS "lp8.arcs"    /* input filename for lp dependencies */
```

Figure 30: Modifications to Application.h

In Figure 30, application.h is set to use eight LPs and use the file, lp8.arcs, to run the simulation. In addition to the arcs file, the map, and spinloop input files had to copied to the HPD directories. The map file is named using the convention: lpX.map where X is equal to the number of LPs. Once these files were set up and the application compiled, SPECTRUM and the coprocessor were ready to be tested.

5.7 Carwash Simulation Results

The carwash was the simplest simulation used to measure the coprocessor's performance. Since the carwash's LPs are fixed, the simulation's granularity was the only factor which varied. The spinloop's value was varied from 1 to 500 in steps of 50. The simulation times from varying the spinloops are shown in Table 8. The results in this table come from running the carwash with the filter, u_null_mess.c, which is based on the conservative Chandy-Misra protocol. Several other filters, called del_null.c and safeclocks.c, which are variations of the Chandy-Misra protocol, were developed at AFIT and were also used to measure the carwash's performance. These filters were so similar to the u_null_mess filter that they did not significantly alter the simulation's performance. Since the other filters had similar results, their times were not included.

Spinloop Size	Avg. Time without Coprocessor (sec)	Avg Time with Coprocessor (sec)	Avg Speedup
1	0.221	0.728	0.304
10	0.615	0.716	0.859
38	0.853	1.636	1.918
50	2.092	0.976	2.143
100	3.930	1.622	2.423
150	5.799	2.303	2.518
200	7.595	2.974	2.554
250	9.435	3.646	2.588
300	11.294	4.393	2.571
350	13.147	5.077	2.590
400	14.937	5.770	2.589
450	16.788	6.426	2.613
500	18.697	7.131	2.622
1000	37.443	13.832	2.707
2000	72.958	27.331	2.669

Table 8: Results From Altering Nullwash's Granularity

The carwash times are plotted in Figure 31 and the speedup is plotted in Figure 32. With 1 spinloop, the simulation ran slower with the coprocessor. This was not surprising because of the extra message passing latencies incurred from the coprocessor's use. When the spinloop was set to 38, the message passing latencies were partially canceled and the coprocessor provided a speedup of 1.98. Figure 32 shows that as the number of spinloops was increased beyond 200, the coprocessor provided a constant speedup of approximately 2.6. This constant acceleration occurs when the spinloop delay is big enough to cancel the message passing latencies between the LPs and CPs. Figure 32 confirms an earlier prediction stated that 190 spinloops would be needed to cancel the message passing latencies between the LPs and CPs in the carwash. The largest test used 2000 spinloops, approximately a 4 millisecond delay for each event. Since the simulation's granularity's are not likely to get much coarser, no larger measurements were taken.

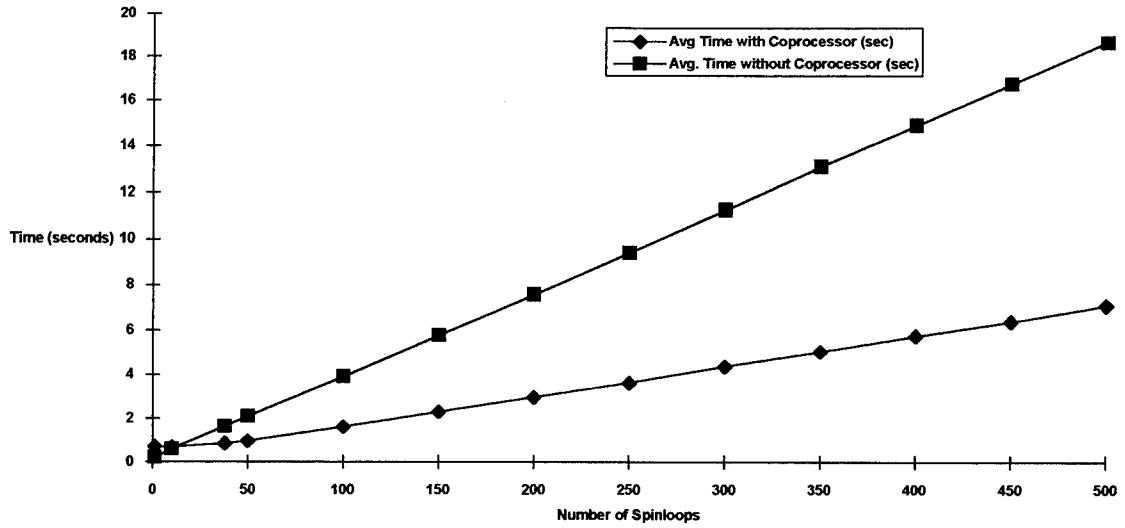

Figure 31: Carwash Times using the Coprocessor on the Null Filter

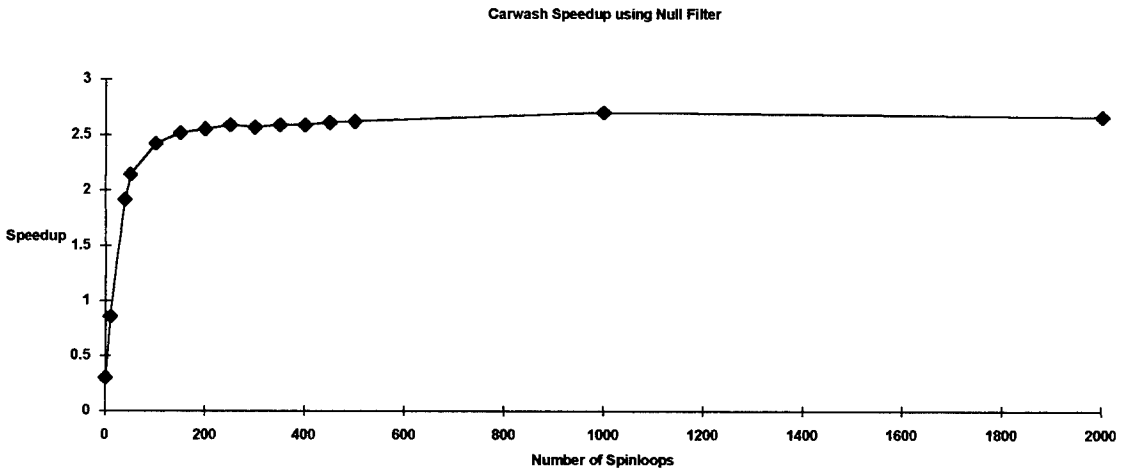

Figure 32: Carwash Speedup using the Coprocessor on the Null Filter

The carwash's performance measurements revealed many characteristics of the parallelism that the coprocessor was designed to exploit. Much of the work on designing a PDES coprocessor at AFIT has emphasized the carwash simulation. This simulation is interesting because it exemplifies how a poorly partitioned simulation performs. Since one of the nodes in the carwash has no input arcs, it races ahead of the other nodes and creates a load imbalance. The

rest of the nodes send a lot of null messages as they catch up and, because the carwash is such a fine grained simulation, the time spent sending and receiving null messages is excessive. One danger of using the carwash simulation with SPECTRUM is that the node without any input arcs can send so many messages that the rest of the LPs spend all of their time receiving these messages. To prevent the LPs from spending all of their time receiving messages, SPECTRUM only receives ten messages at a time before it goes back to processing events. Since the coprocessor is designed to send and receive messages for the LP for which it works, it effectively eliminates the time that the LP spends sending and receiving null messages. The coprocessor can receive messages, queue them in its NEQ, and keep track of the minimum times on the input arcs while the LP continues to do useful work. Since the carwash sends so many null messages, the coprocessor provides a significant amount of speedup. On more evenly partitioned simulations, such as the Wallace tree multiplier, the coprocessor is less likely to accelerate the simulation.

5.8 Wallace Tree Multiplier Results

The Wallace tree multiplier provided a more realistic test for the coprocessor. It was run using three different partitioning strategies and the number of LPs used in the simulation was varied from 2 to 32. To approximate the simulation's sequential time, the simulation was also run using one LP. Varying these factors and the number of spinloops meant that a large of number of simulations had to be run. At least five simulations were run for each configuration to minimize the variance of the results.

Because all of the partitioning strategies tested attempt to place an equal number of gates on the LPs, the Wallace tree simulations tended to be more balanced than the carwash. An examination of a Wallace tree multiplier schematic confirmed that the Wallace tree multiplier

does not contain significantly large feedback cycles. The Wallace tree multiplier consists of adders connected in a tree in which signals flow straight through the adders to the outputs. This lack of feedback was expected since the Wallace tree multiplier performs a fixed function. The absence of feedback improves the likelihood that the no feedback partitions will perform well.

5.8.1 Random Partitioning

Random partitioning was the first partitioning strategy tested. Since this strategy randomly placed circuit elements on LPs, it introduced feedback into the design and increased the number of arcs between LPs. The times for the randomly partitioned Wallace tree are shown in Table 9. In addition to the Wallace tree's times, Table 9 also contains approximations for ideal speedup and parallel overhead. These are calculated using Equation 4 and Equation 5. A problem with VSIM's filter prevented the randomly partitioned simulation from being run on 22 LPs. This problem results from an overflow in the time, in TIME_FILTER, when the simulation clock was advanced. Hurford experienced the same problem in his research on the iPSC/2, but with a different number of LPs. Because of this problem, no results were obtained for random partitions using 22 LPs.

The results in table 9 imply that the coprocessor introduces a large amount of overhead for any number of LPs. The large overhead resulting from the coprocessor's use is probably exaggerated by the large message passing delays between the LPs and their CPs. This large overhead can be seen by comparing Figure 33 and Figure 34 and is especially pronounced for small numbers of LPs.

# of LPs	Avg Times w/o CP (sec)	Time w/Ideal Speedup (sec)	Overhead (sec)	Avg Times w/CP (sec)	Time w/Ideal Speedup (sec)	Overhead (sec)
1	10.414			**12.898**		
2	5.877	5.207	0.670	16.422	6.449	10.072
4	**4.660**	**2.604**	**2.057**	16.521	3.225	13.297
6	4.676	1.736	2.940	14.088	2.150	11.939
8	5.031	1.302	3.729	14.310	1.612	12.698
10	5.699	1.041	4.658	14.272	1.290	12.982
12	6.590	0.868	5.722	14.179	1.075	13.104
14	7.671	0.744	6.927	13.756	0.921	12.835
16	8.896	0.651	8.245	15.658	0.806	14.852
18	10.332	0.579	9.753	14.941	0.717	14.224
20	11.906	0.521	11.385	15.900	0.645	15.255
24	14.377	0.434	13.943	18.939	0.537	18.402
26	16.378	0.401	15.977	17.004	0.496	16.508
28	18.241	0.371	17.870	18.180	0.461	17.719
30	19.631	0.347	19.284	20.920	0.430	20.490
32	20.259	0.325	19.934	18.607	0.403	18.204

Table 9: Wallace Tree Multiplier Results Using Random Partition and 1 Spinloop

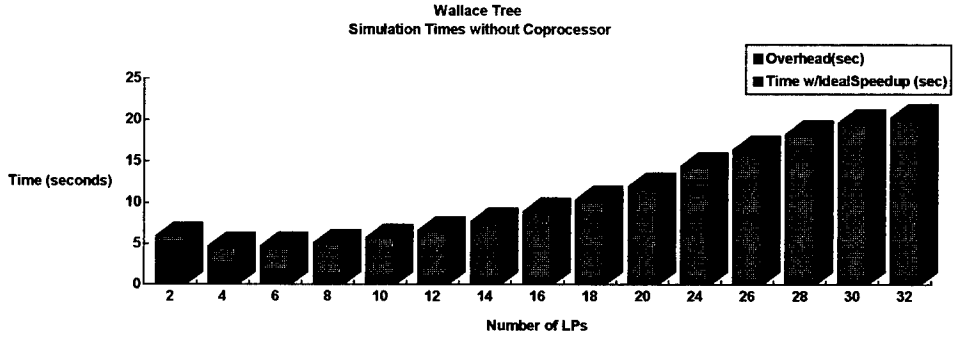

Figure 33: Wallace Tree Times without Coprocessor, using Random Partition and 1 Spinloop

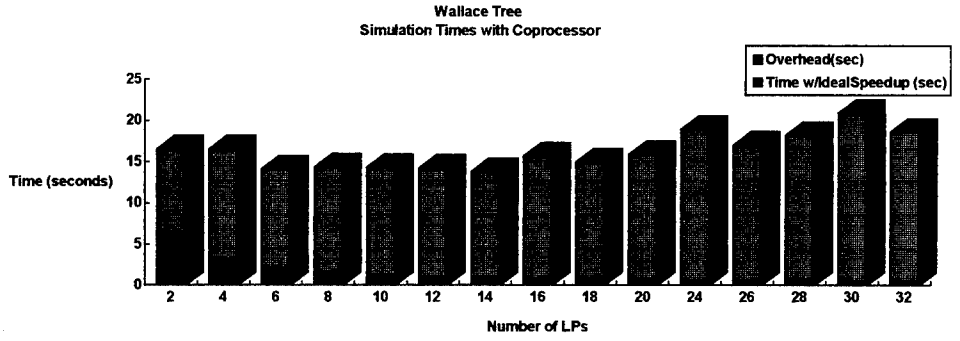

Figure 34: Wallace Tree Times with Coprocessor, using Random Partition and 1 Spinloop

Even though the coprocessor did not accelerate the randomly partitioned Wallace tree, Figures 33 and 34 illustrate one of the coprocessor's advantages: the coprocessor reduces the growth in parallel overhead as the simulation uses more LPs. When the Wallace tree is partitioned among more LPs, the resulting LPs have more arcs connecting them to other LPs. Since the coprocessor handles messages sent and received along these arcs while the LP continues to perform useful work, the simulation can be accelerated. Unfortunately, this acceleration can occur at a point in which the parallel overhead is greater than the sequential simulation.

To determine how the message passing delay affected the Wallace tree's performance with the coprocessor, the spinloops were increased to 50. The delay caused by spinloops of this size was expected to offset most of the message passing overhead. The results from increasing the spinloop are shown in Table 10 and are graphed in Figure 35 and Figure 36.

# of LPs	Avg Times w/o CP (sec)	Time w/Ideal Speedup (sec)	Overhead (sec)	Avg Times w/CP (sec)	Time w/Ideal Speedup (sec)	Overhead (sec)
1	128.779			131.544		
2	72.314	64.390	7.925	84.578	65.772	18.806
4	42.650	32.195	10.455	56.451	32.886	23.565
6	34.060	21.463	12.597	45.824	21.924	23.900
8	28.451	16.097	12.354	39.238	16.443	22.795
10	26.672	12.878	13.794	36.925	13.154	23.771
12	23.971	10.732	13.239	33.164	10.962	22.202
14	22.270	9.199	13.072	30.390	9.396	20.994
16	**22.159**	**8.049**	14.110	29.964	8.222	21.743
18	22.927	7.154	15.773	29.489	7.308	22.181
20	23.356	6.439	16.917	29.090	6.577	22.513
24	24.189	5.366	18.823	28.960	5.481	23.479
26	25.004	4.953	20.051	28.388	5.059	23.329
28	26.235	4.599	21.636	28.309	4.698	23.611
30	26.383	4.293	22.090	28.492	4.385	24.107
32	26.691	4.024	22.667	**27.696**	**4.111**	**23.585**

Table 10: Wallace Tree Multiplier Results Using Random Partition and 50 Spinloops

Although the increased number of spinloops did not cause the coprocessor to accelerate the simulation, they did show that message passing latencies had a large effect on the coprocessor.

Figure 36 shows that the parallel overhead no longer consumed most of the simulation time when the spinloops were increased. These results also show how the coprocessor could make a simulation more scalable. When the coprocessor was not used, the parallel overhead increased more rapidly than when it was used, but still increased less than it did with a single spinloop. Since the only difference between the randomly partitioned simulations was the rate at which they passed messages, this less-rapid increase in parallel overhead implied that contention was a problem for the randomly partitioned Wallace tree when it only uses one spinloop. This contention will be discussed further in the Wallace tree conclusions.

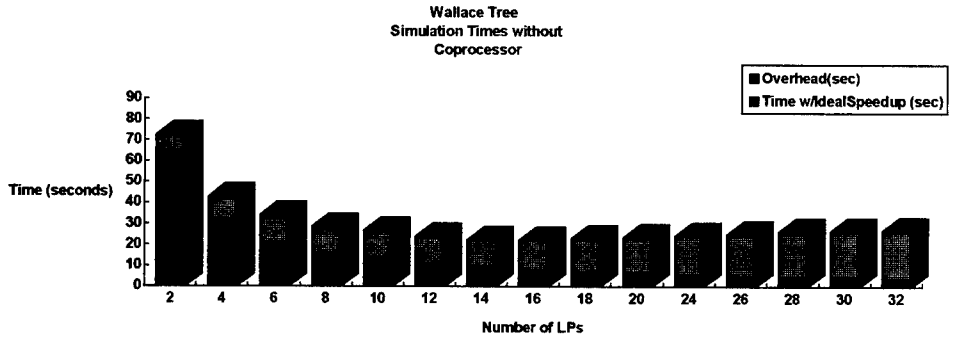

Figure 35: Wallace Tree Times without Coprocessor, using Random Partition and 50 Spinloops

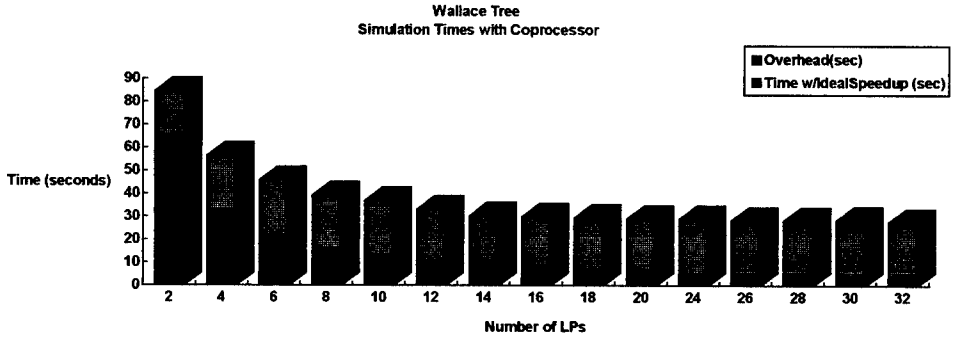

Figure 36: Wallace Tree Times with Coprocessor, using Random Partition and 50 Spinloops

Since the randomly partitioned Wallace tree has many arcs between nodes and passes a lot of null messages, it appears that the coprocessor should accelerate the simulation. The

coprocessor's lack of acceleration indicates that null messages are not driving the simulation's performance. If the LPs spent a majority of their time processing null messages, the coprocessor would probably accelerate the simulation. Instead, the randomly partitioned Wallace tree's performance is limited by two other factors. One factor, the lookahead ratio, is obvious from the way the circuit was partitioned. Since the LPs were constructed by randomly grouping gates, the lookahead is probably only a single gate delay for most arcs between LPs. If the largest arc time was a single gate in the whole system, the simulation would be forced to execute in a time-driver manner and the simulation would be communications-bound. Another factor, related to lookahead, that is slowing the simulation is that the LPs have to wait on their slowest input arc to catch up to the LP's local time before the LP can process any events. If one of an LP's input arcs consistently lags behind the LP's other arcs, the LP will spend a large portion of its time in a blocked state, waiting for events to arrive on that arc. When an LP is blocked, it can no longer execute events in parallel with the coprocessor and the advantage of the coprocessor is lost. Since the randomly partitioned Wallace tree has a small lookahead ratio and since its LPs spend a large portion of their time in a blocked state, the coprocessor does not accelerate the simulation.

5.8.2 Breadth-First, No Feedback Partitioning

The breadth first, no feedback, partitioning strategy is an improvement over random partitioning for several reasons. One improvement lies in the fact that the LPs consist of strongly connected components and removes any partition-induced feedback arcs that were caused by the random partitioning algorithm. The removal of feedback arcs helps increase lookahead and reduce the amount of time that the LPs have to block. The breadth-first, no feedback partition's run times shown in Table 11 reflect these improvements over random partitioning.

# of LPs	Avg Times w/o CP (sec)	Time w/Ideal Speedup (sec)	Overhead (sec)	Avg Times w/CP (sec)	Time w/Ideal Speedup (sec)	Overhead (sec)
1	10.387			12.850		
2	5.912	5.194	0.719	8.634	6.425	2.209
4	3.354	2.597	0.757	6.209	3.213	2.997
6	2.487	1.731	0.756	4.905	2.142	2.763
8	2.246	1.298	0.948	4.523	1.606	2.917
10	2.826	1.039	1.787	4.374	1.285	3.089
12	2.156	0.866	1.290	4.095	1.071	3.024
14	2.705	0.742	1.963	**3.767**	**0.918**	**2.849**
16	2.265	0.649	1.616	3.811	0.803	3.008
18	2.249	0.577	1.672	3.939	0.714	3.225
20	**1.994**	**0.519**	**1.475**	4.177	0.643	3.535
22	2.401	0.472	1.929	4.356	0.584	3.772
24	2.228	0.433	1.795	4.580	0.535	4.045
26	2.055	0.400	1.656	4.529	0.494	4.035
28	2.074	0.371	1.703	4.515	0.459	4.056
30	2.996	0.346	2.650	4.863	0.428	4.435
32	2.788	0.325	2.463	4.970	0.402	4.568

Table 11: Wallace Tree Multiplier Results Using Breadth-First Partition and 1 Spinloop

The results in Table 11 show a large improvement over the random partition's times shown in Table 10. Even with the spinloops set to 1, the simulation's parallel overhead is smaller than it was for the random partition. The coprocessor was affected by the improved partition in several ways. Since the breadth-first, no feedback partition reduced the time that the LP spent in a blocked state, it reduced the coprocessor's parallel overhead. The reduction of the blocked times decreased the number of null messages needed. The breadth-first, no feedback partition also reduced the number of arcs connecting the LPs and in doing so, lowered the number of nulls even further. The reduction of parallel overhead enhanced the coprocessor's performance, but the reduction of null messages also significantly limited the coprocessor's ability to work in parallel with the LPs. Figure 37 and Figure 38 illustrate the lower parallel overhead from using the breadth-first, no feedback partition.

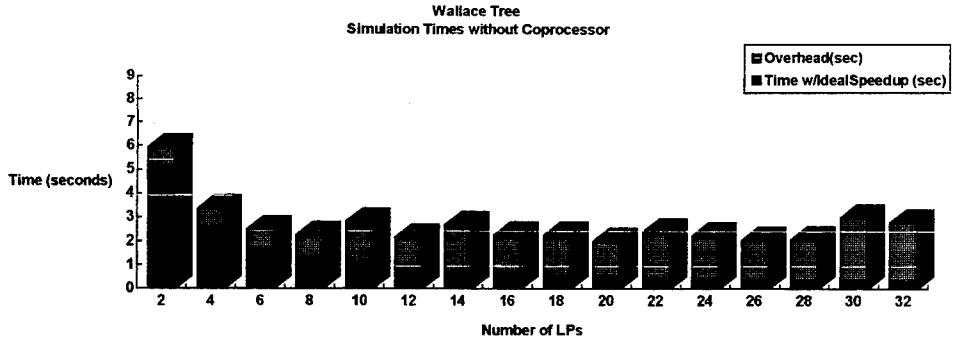

Figure 37: Wallace Tree Times without Coprocessor, using Breadth-First Partition and 1 Spinloop

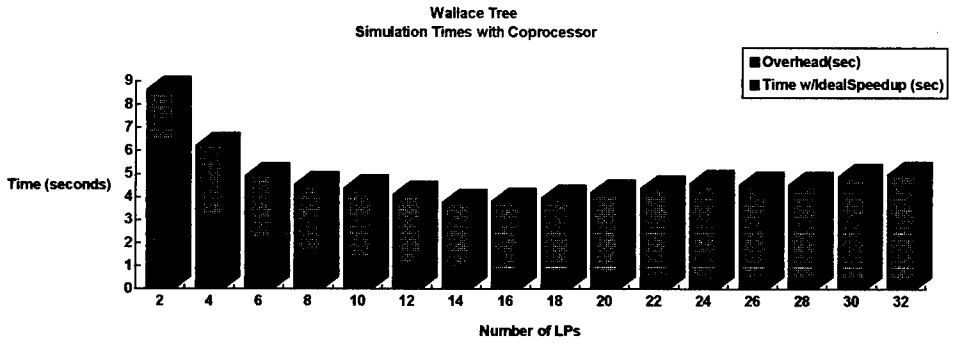

Figure 38: Wallace Tree Times with Coprocessor, using Breadth-First Partition and 1 Spinloop

5.8.3 Breadth-by-Source, No Feedback Partitioning

Breadth-by-source was the last partitioning method tested using the Wallace tree multiplier. This method is similar to breadth-first-search except that it tries to distribute the sources equally among all of the LPs. This distribution of sources might improve the simulation's performance if having the source allows an LP to constantly execute events, but it also might hurt the simulation's performance of it causes one of the input arcs to constantly stay ahead of the other input arcs. It would probably be better to place all of the sources on LPs in a manner which minimized the differences in the times on each of the LP's input arcs. The timing measurements for the breadth-by-source strategy are shown in Table 12.

# of LPs	Avg Times w/o CP (sec)	Time w/Ideal Speedup (sec)	Overhead (sec)	Avg Times w/CP (sec)	Time w/Ideal Speedup (sec)	Overhead (sec)
1	10.456			12.837		
2	5.574	5.228	0.346	8.787	6.419	2.460
4	2.901	2.614	0.287	5.498	3.209	2.289
6	2.690	1.743	0.947	4.662	2.140	2.523
8	2.585	1.307	1.278	4.293	1.604	2.688
10	3.425	1.046	2.379	3.929	1.284	2.645
12	4.370	0.871	3.499	**3.633**	**1.070**	**2.563**
14	2.485	0.747	1.738	3.773	0.917	2.856
16	3.460	0.654	2.807	3.753	0.802	2.951
18	2.586	0.581	2.005	3.855	0.713	3.142
20	2.514	0.523	1.991	3.765	0.642	3.123
22	**2.495**	**0.475**	**2.020**	4.054	0.584	3.471
24	3.555	0.436	3.119	4.285	0.535	3.750
26	3.147	0.402	2.745	4.584	0.494	4.090
28	3.079	0.373	2.706	4.847	0.458	4.389
30	3.108	0.349	2.759	5.151	0.428	4.723
32	3.227	0.327	2.900	5.254	0.401	4.853

Table 12: Wallace Tree Multiplier Results Using Breadth-by-Source Partition and 1 Spinloop

Without the coprocessor, the breadth-by-source partitioning strategy was faster than the random strategy, but was slower than the breadth-first-search with no feedback. Moving the sources probably slowed the simulation because the sources caused some input arcs to get ahead of others. When the coprocessor was used, breadth-by-source was slightly faster than breadth-first because the coprocessor could reduce the effects of one input arc in an LP preceding others. The execution times for the breadth-by-source partition are in Figure 39 and Figure 40. These figures indicate that the coprocessor incurs much more parallel overhead for small numbers of nodes than for simulations that don't use the coprocessor. To determine if this overhead could be reduced by increasing the simulation's granularity, the spinloops were increased to 50.

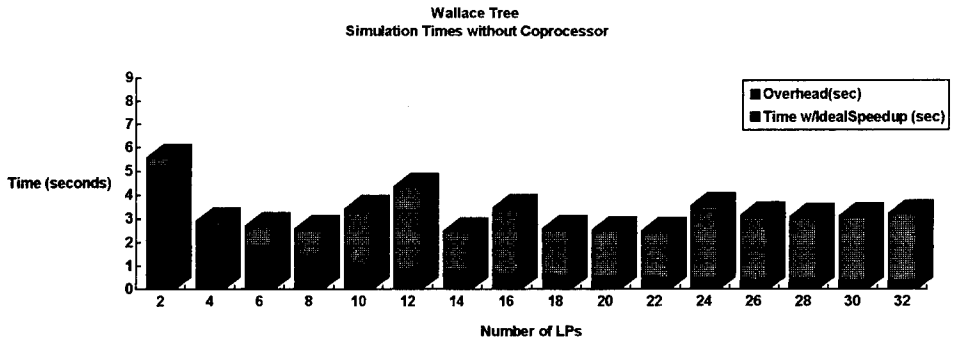

Figure 39: Wallace Tree Times without Coprocessor, using Breadth-by-Source Partition and 1 Spinloop

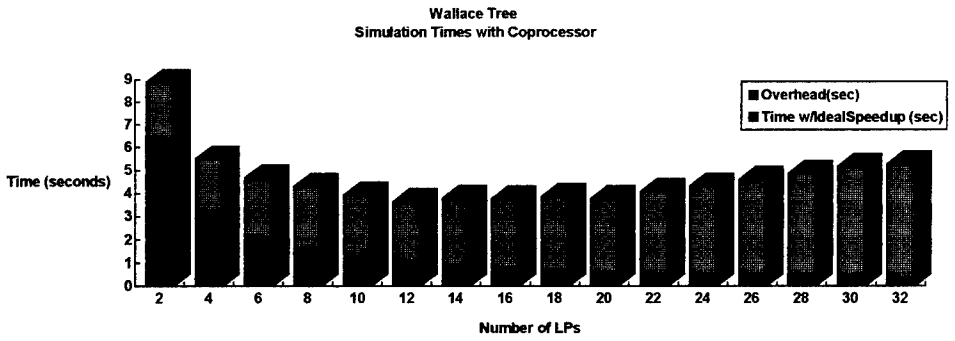

Figure 40: Wallace Tree Times without Coprocessor, using Breadth-by-Source Partition and 1 Spinloop

Increasing the Wallace tree's spinloops to 50 compensated for much of the penalty of LPs sending messages to coprocessors on a separate node. The times for the coprocessor shown in Table 13 compare more favorably to the normal runs than they did in Table 12.

# of LPs	Avg Times w/o CP (sec)	Time w/Ideal Speedup (sec)	Overhead (sec)	Avg Times w/CP (sec)	Time w/Ideal Speedup (sec)	Overhead (sec)
1	129.519			130.369		
2	84.958	64.760	20.199	85.785	65.185	20.601
4	43.627	32.380	11.247	46.029	32.592	13.437
6	34.484	21.587	12.898	36.749	21.728	15.021
8	30.076	16.190	13.886	32.269	16.296	15.973
10	23.337	12.952	10.385	25.416	13.037	12.379
12	19.170	10.793	8.377	20.676	10.864	9.812
14	23.209	9.251	13.958	24.776	9.312	15.464
16	22.916	8.095	14.821	24.234	8.148	16.086
18	22.458	7.196	15.263	24.037	7.243	16.794
20	19.668	6.476	13.192	21.441	6.518	14.923
22	16.454	5.887	10.567	18.438	5.926	12.512
24	14.680	5.397	9.283	16.480	5.432	11.048
26	17.879	4.982	12.898	19.181	5.014	14.167
28	16.091	4.626	11.465	17.403	4.656	12.747
30	15.551	4.317	11.234	16.628	4.346	12.282
32	**14.360**	**4.047**	**10.313**	**15.754**	**4.074**	**11.680**

Table 13: Wallace Tree Multiplier Results Using Breadth-by-Source Partition and 50 Spinloops

The breadth-by-source Wallace tree simulation with 50 spinloops is still faster than the randomly partitioned simulation with the same number of spinloops when they are increased to 50, but the difference is less noticeable than before the spinloops were added. The increased granularity appears to reduce contention and greatly improves the simulation's ability to be scaled up. This effect of granularity will be further discussed in the Wallace tree conclusions. Increasing the granularity also reduced the parallel overhead, as shown in Figure 41 and Figure 42.

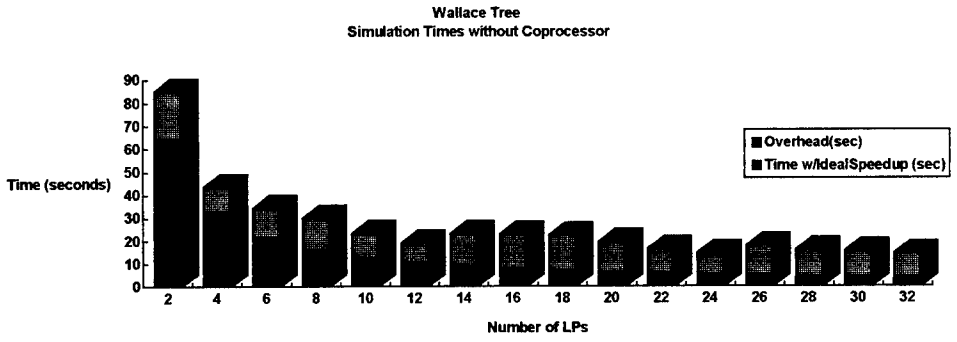

Figure 41: Wallace Tree Times without Coprocessor, using Breadth-by-Source Partition and 50 Spinloops

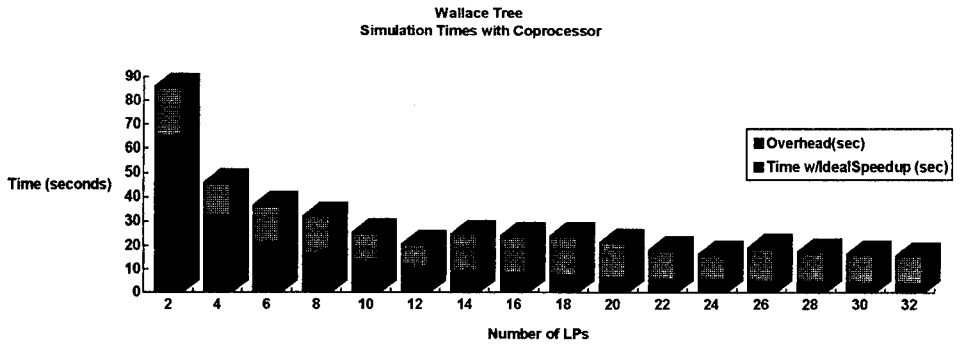

Figure 42: Wallace Tree Times with Coprocessor, using Breadth-by-Source Partition and 50 Spinloops

5.8.4 Wallace Tree Multiplier Conclusions

The Wallace tree multiplier was useful for testing the coprocessor because it represented a large combinatoric circuit used to solve a simple problem. Since it had no feedback, the partitioning strategies that avoided feedback loops provided a large amount of acceleration. The efficient partitioning strategies accelerated the Wallace tree more than the coprocessor did because the coprocessor did not reduce the amount of time that the LPs spent blocking. The Wallace tree simulation times with 1 spinloop are summarized in Figure 43. From this figure, it is clear that the best partitioning strategy tested for the Wallace tree is breadth-first with no feedback and no coprocessor.

Figure 44 summarizes the Wallace tree simulation times with 50 spinloops. Changing the Wallace tree multiplier's granularity had a profound effect on the random partition's performance because it reduced the increase in parallel overhead as LPs were added to the simulation. This can be seen by comparing Figure 43 with Figure 44. In fact, the parallel overhead was reduced to the point where the breadth-by-source partition has less of an advantage over the random partition than it did with one spinloop. The random partition's improvement is not surprising because, by adding a uniform, constant delay to all of the LPs without increasing the number of messages sent, the spinloop is similar to a perfect partition. As the time spent in the spinloop dominates the time

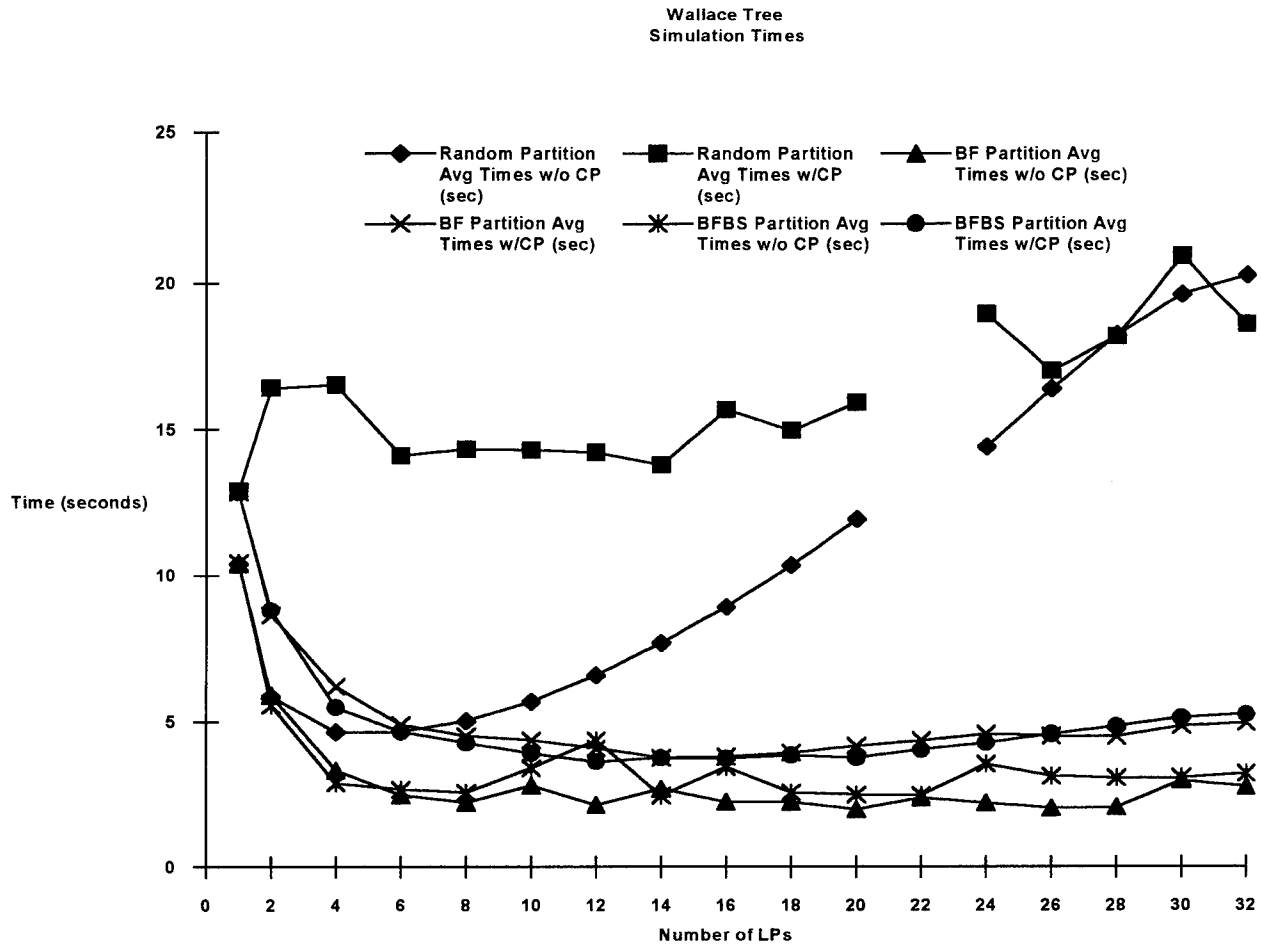

Figure 43: Wallace Tree Performance Summary with 1 Spinloop

spent running the simulation, the simulation will obtain a close to ideal speedup when additional nodes are added. The spinloops are useful for altering a simulation's granularity, but are not useful for eliminating the effects of message passing latencies between the LPs and CPs if the number of spinloops is set too high. The additional computation time incurred from the use of the spinloop should be equal to the time an LP spends passing messages between itself and its LP.

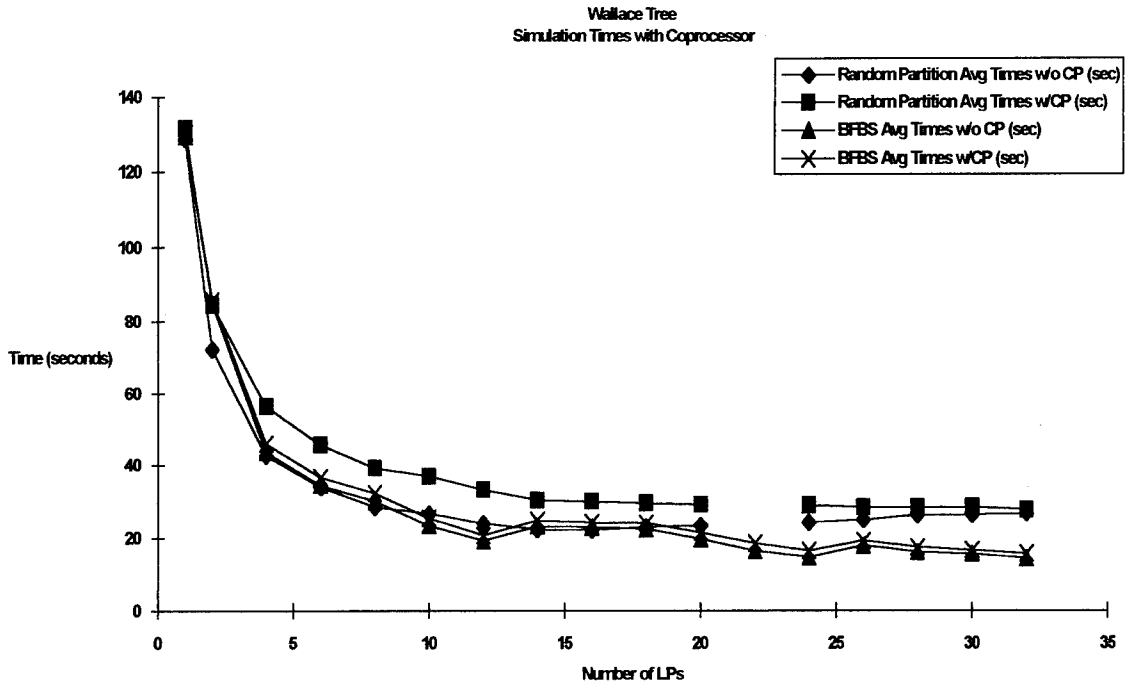

Figure 44: Wallace Tree Performance Summary with 50 Spinloops

5.9 Associative Memory Results

The associative memory simulation differs from the Wallace tree multiplier in several significant ways. The associative memory does not perform a simple operation on the data input to it. Instead, it stores and recalls data and therefore contains many internal feedback loops. Since the associative memory was much larger than the Wallace tree, its execution times were much larger than previous simulations. Table 14 shows the results of running it using the random partition with one spinloop. Each average time in Table 14 represents the average of at least five samples.

# of LPs	Avg Times w/o CP (sec)	Time w/Ideal Speedup (sec)	Overhead (sec)	Avg Times w/CP (sec)	Time w/Ideal Speedup (sec)	Overhead (sec)
1	1313.908			1319.565		
2	516.171	656.954	-140.783	870.127	659.783	210.345
4	230.754	328.477	-97.723	659.521	329.891	329.630
6	142.037	218.985	-76.948	464.075	219.928	244.148
8	105.762	164.239	-58.477	355.464	164.946	190.518
10	95.426	131.391	-35.965	306.446	131.957	174.490
12	87.540	109.492	-21.952	262.267	109.964	152.303
14	86.008	93.851	-7.843	240.687	94.255	146.432
16	**85.673**	**82.119**	**3.554**	216.233	82.473	133.760
18	92.092	72.995	19.097	209.538	73.309	136.229
20	95.489	65.695	29.794	193.776	65.978	127.798
22	98.091	59.723	38.368	**181.588**	**59.980**	**121.608**
24	107.201	54.746	52.455	185.369	54.982	130.387
26	119.862	50.535	69.327	193.162	50.753	142.410
28	126.948	46.925	80.023	184.126	47.127	136.999
30	138.850	43.797	95.053	189.643	43.986	145.658
32	154.973	41.060	113.913	197.655	41.236	156.419

Table 14: Associative Memory Results Using Random Partition and 1 Spinloop

The associative memory simulations provided some surprising results. The negative overhead values for the simulation imply a super-linear speedup, but really indicate that the sequential time being compared to the parallel times is not optimal. There exists some faster algorithm that would reduce the sequential time. Part of the slow sequential results may be a result of cache misses or some other delay in the Paragon's memory hierarchy, but is mainly a result of the way VSIM inserts events into its next event queue. VSIM's queue is implemented with a linked list. When it inserts an event into its queue, it starts from the head of the queue and traverses the linked list until it finds the appropriate location for the event. Since VSIM starts from the head of its queue, it has to traverse the entire linked list to insert an event scheduled later than all the other events in the queue. When one LP is used, the LP will have to traverse a much longer linked list and most of the events will be inserted at the tail of the queue. As additional LP's are used to run the simulation, the size of the queues decrease and inserting events into them

becomes more efficient. If VSIM was modified to insert events by starting at the end of the queue, the sequential simulation would run much faster.

The coprocessor increased the associative memory's message passing overhead because its use increased the amount of time the LPs had to block. Contention probably also contributed to the poor speedup since the coprocessor increases the number of messages passed in a simulation which is already communications bound.

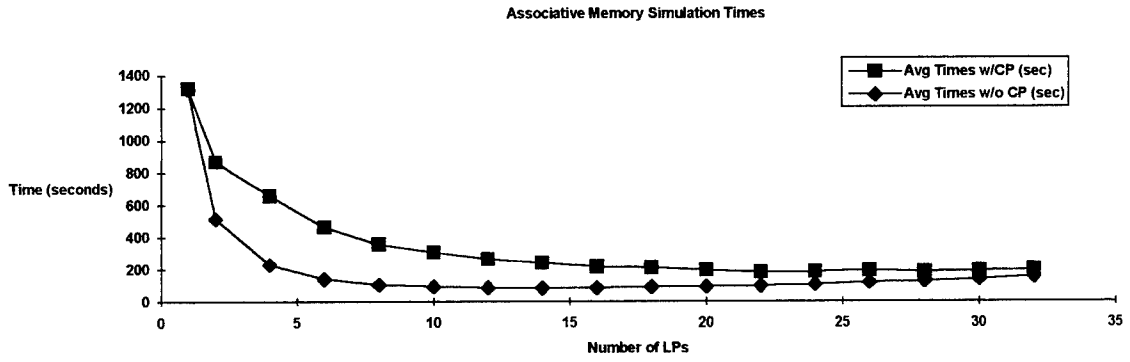

Figure 45: Associative Memory Times with 1 Spinloop

5.10 Conclusion

The coprocessor did not provide any acceleration for most of the simulations. It did accelerate the carwash when spinloops were used to increase its granularity, but most of the other simulations were slowed considerably by the message passing latencies between the coprocessors and the LPs. Although the coprocessor did not accelerate the simulations, it did provide several important conclusions about ways to accelerate parallel discrete event simulations. These insights are described in the next chapter.

VI. Conclusions

6.1 Introduction

This chapter describes the conclusions drawn from this research and gives recommendations for future research. The conclusions drawn pertain to both the coprocessor and parallel simulations in general.

6.2 Conclusions

Much of the previous research at AFIT concentrated on reducing the amount of null messages necessary to maintain the simulation's causality. The coprocessor's main design goals were to offload the work of sending and receiving null messages and the work needed to manage the simulation's NEQ. Since most of the VHDL simulations were not accelerated by the use of the coprocessor, it important to reexamine the role null messages play in determining the simulations' run times. Null messages can slow a simulation by causing contention on the parallel computer's communication channels and can dominate the LP's workload, but they are not the simulation's real problem. Excessive null message overhead is a symptom of a non-ideal partition in which the LPs are blocked for long periods of time. The amount of time that the LPs block can be minimized by maximizing the LPs' lookahead ratios and partitioning the LPs so that channel times on all of the LPs' input arcs progress consistently. When an input arc falls behind the other input arcs on an LP, the LP must block until that arc catches up. This blocking time is only exacerbated by the presence of feedback loops. Therefore, minimizing the number of feedback loops will generally improve the simulation's performance.

Partitioning the circuits so that the sources are evenly distributed among the LPs can cause some of an LP's input arcs to get ahead of the rest of the simulation. If sources which generate events at approximately the same rate are placed on the same LP, that LP is likely to run faster because there is less chance of it having to wait on the sources. This research confirmed that partitioning by sources can be slower than simple feedback loop elimination.

The coprocessor demonstrated its ability to accelerate poorly partitioned simulations with the carwash. The coprocessor was able to reduce the effects of the carwash's load imbalance and fine granularity. The coprocessor might be able to provide a reasonable amount of speedup in BATTLESIM applications because realistic battle simulations tend to get unbalanced. Dynamically partitioned simulations would also probably benefit from a coprocessor similar to the one developed in the research. The coprocessor could minimize the effects of small load imbalances and excessive nulls being generated by a small number of LPs. If such a coprocessor was expanded to perform some of the load balancing tasks, it would provide even more speedup.

VHDL circuits are static by nature and can be partitioned relatively efficiently. Since the VSIM simulations tended to be more evenly partitioned, the coprocessor did not accelerate them. The application level NEQ used in VSIM also limited the amount of lookahead the LPs were able to exploit. Because the VSIM NEQ did not send nulls to update its LP's output arcs, the only opportunity for lookahead occurred when an LP received an event from SPECTRUM's NEQ. Since most of the events were executed from VSIM's NEQ, most of the opportunities to exploit lookahead were missed. VHDL simulations can be accelerated using better partitioning strategies. The simulation's algorithm plays such a large role in determining the simulation's run time that any significant improvements to the algorithm will result in a far better speedup than a corresponding improvement in the hardware.

6.3 Recommendations

Since BATTLESIM simulations are less likely to be partitioned efficiently than the VHDL simulations, the coprocessor should be tested with BATTLESIM. Porting BATTLESIM to the Paragon should be fairly simple. BATTLESIM's makefiles would need to be modified, but few other changes would be required.

Another future area of research would be to see how the coprocessor could assist a simulation that used dynamic arc times. The use of dynamic arc times would allow the application to calculate lookahead when it sent null messages. This would probably increase the simulation's lookahead ratio and improve the simulation's performance. Dynamic lookahead could provide a large amount of speedup in VSIM simulations when the LP was executing events out of VSIM's application level queue. The lookahead would improve the simulation's performance the most when the simulation was executing events for gates that were logically close to the LP's input arcs. Since these gates are the farthest from the LP's outputs, it would be simple to calculate how many gate delays the signal would take, using the shortest possible path, before the events at the gate could affect the output. Such calculations would require the application to have some knowledge about what gates were on the LP, but this information could easily be passed from SPECTRUM's node level. The partitioning tool would have performed many of the calculations when it divided the gates among the LPs and the results could be placed in an expanded map file.

Any future effort to try to minimize the time differences on the LP's input arcs would be best served by allowing the partitioning strategy to know information about the simulation's inputs. By knowing the average rate at which various inputs changed, the partitioning tool could place sources with the same event rate on the same LP. Knowing simulation specific information

such as source arrival rates would allow the partitioning tool to use queuing theory to efficiently partition the rest of the circuit. Such partitions would probably minimize the amount of time that the LPs were blocked and could provide large amounts of acceleration.

6.4 Summary

The coprocessor provided many insights as to how parallel discrete events can be accelerated. It was able to accelerate poorly partitioned, fine-grained simulations such as the carwash, but did not accelerate balanced simulations. For most evenly balanced simulations, better acceleration can be obtained by improving the simulation's partition than by designing a hardware coprocessor. The coprocessor provides the most benefits when the partitioning strategy cannot be improved. Future efforts in accelerating parallel simulations should include examining ways to maximize the LPs' lookahead ratio and finding ways to minimize the time differences on the LPs' input arcs.

Appendix A. Concerns For the Future Use of SPECTRUM Simulations

This Appendix describes several issues which might affect future users of SPECTRUM. The comments in this appendix apply to using SPECTRUM on both the iPSC/2 and the Paragon unless otherwise noted.

Carwash Memory Leaks

The carwash simulation contains a significant memory leak that could affect its performance if the simulation was lengthened. When an application calls lp_post_event( ), it passes the function an event to be posted. This event, which is dynamically allocated, is either posted to the LP's next event queue or sent to the appropriate LP. SPECTRUM makes a copy of this event when it processes it so that the application cannot release its memory. SPECTRUM assumes that the original event will be freed by the application when the application no longer needs it. In the carwash simulation, old events that have been posted are never eliminated. Instead, the pointer to the events is changed and the events continue to consume memory as the LP processes them. If the carwash was run long enough, it would eventually use all of a node's memory and would crash the iPSC/2. On the Paragon, the memory leaks would eventually cause the Paragon to page virtual memory to and from its hard drives, and would cause the simulation's performance to degrade.

Problems with VSIM's Filter

This research observed two problems with VSIM's filter. There are several type mismatches in the filter and the filter cannot prevent large clock values from causing the clock

variable to overflow. The type mismatches occur because VSIM does not include the header file, vsim.h when the filter is compiled. In this header file the function, get_low_time( ), is prototyped to return an unsigned integer. Since the header is not compiled with the filter, the C compiler assumes that get_low_time( ) returns an integer. Thus in the application level, get_low_time( ) is expected to return a variable with a different type than the variable returned in the filter. Fortunately, the unsigned integer's representation is close enough to the integer's that the simulation is not affected. This type mismatch still might cause problems if SPECTRUM or the filter were modified.

The second problem with VSIM's filter was more serious. During long simulations the clock time can exceed the size that the variable, my_clock can hold. When the LPs call lp_advance_time( ) in such a case, my_clock overflows and goes from a very large value to a very small value. This small value is incorrectly recognized as a causality error and causes the simulation to abort. The clock can overflow even when the end of the simulation's end-time is set lower than the maximum value that my_clock can hold because of the way VSIM terminates. The clock overflowed on several partitions in this research's test cases and prevented data collection from those partitions. Hurford noted in his research that VSIM's filter exhibited the same problem on the iPSC/2[25].

Using the Paragon's Interactive Parallel Debugger with CPSPECTRUM

CPSPECTRUM, since it changes the application's process group when it loads the coprocessors, will not normally work with the Paragon's Interactive Parallel Debugger (IPD). To allow IPD to be used with CPSPECTRUM, a conditional compile switch called "IPD" has been added. This switch, when defined, alters CPSPECTRUM so that it will not automatically load the

coprocessors. This means that the coprocessors have to be loaded manually from either the command line or IPD's prompt.

The IPD flag is set in the makefile by adding "-DIPD" to the compile switches as shown in the following line:

```
PARA_CFLAGS=-c -g -DIPD -DBUSY -UOUTPUT -DCOUNTS -UMONITORCUBE -UMAPPING
```

When the IPD compile switch is defined, the carry lookahead adder (CLA) simulation and its coprocessors can be loaded manually from the Paragon's command line with the following command:

```
cla 150 -on 0,2,4,6,8,10,12,14 -sz 16 \; vsim_cp -on 1,3,5,7,9,11,13,15
```

This command runs the simulation for 150 ns on 16 nodes. The "\" is used to inform the Paragon that the executables cla and vsim_cp are to execute in the same process group and is necessary for the coprocessors to work with the LPs. The LPs are loaded on even-numbered nodes and the CPs are loaded on odd-numbered nodes.

The CLA simulation and coprocessors can be loaded at IPD's prompt with the following command:

```
load cla 150 -sz 16 -on 0,2,4,6,8,10,12,14\; vsim_cp -on 1,3,5,7,9,11,13,15
```

This command works exactly the same way that the previous command did for the Paragon's command line.

Both of the commands for manual CP loading have been tested with revision 1.2 of the Paragon's operating system. The Paragon command line interpreter has trouble when arguments are passed to a parallel application. Both of these commands, since they pass arguments to the application, work only in the order shown. If the application argument or the -sz switch is moved, the command will no longer work. The order of the arguments in these commands contradicts the

order shown in the Paragon's on-line manual pages. The Paragon's on-line pages suggest ordering the commands in a different order, but that order does not work. Future versions of the Paragon's operating system will probably improve the way it handles command line arguments.

[1] Hennessy, John L. and David A. Patterson. *Computer Architecture: A Quantitative Approach.* San Mateo: Morgan Kaufmann Publishers, Inc. 1990.

[2] Fujimoto, Richard M. *Parallel Discrete Event Simulation.* School of Information and Computer Science, Georgia Institute of Technology. Atlanta, Georgia 30332.

[3] Chandy, K. M. and J. Misra. "Asychronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the Association of Computing Machinery.* 20(1):198-206 (April 1981).

[4] Daniel, David W. *Design of a Hardware Discrete Event Simulation Coprocessor.* MS Thesis, AFIT/GCE/ENG/93M-01, Air Force Institute of Technology (AU) Wright-Patterson AFB, OH, March 1993.

[5] Berlin, Jacob L. *Design of a Parallel Discrete Event Simulation Coprocessor.* MS Thesis, AFIT/GCS/ENG/93D-02, Air Force Institute of Technology (AU) Wright-Patterson AFB, OH, December 1993.

[6] Reynolds, Jr. Paul F. et. al. "Comparative Analysis of Parallel Simulation Protocols," *Proceedings of the 1989 Winter Simulation Conference.* 671-679. 1989.

[7] Fujimoto, Richard M. "Parallel Discrete Event Simulation." *Communications of the Association of Computing Machinery.* 33(10):31-53 (October 1990).

[8] Misra, J. "Distributed Discrete-Event Simulation," *Computing Surveys*, the Association of Computing Machinery. 18(1):39-65 (March 1986).

[9] Righter, Rhonda and Jean C. Walrand. "Distributed Simulation of Discrete Event Systems," *Proceedings of the IEEE.* 77(1):99-113 (January 1989).

[10] Fujimoto, Richard M., Jya-Jang Tsai and Ganesh C. Gopalakrishnan. "Design and Evaluation of the Rollback Chip: Special Purpose Hardware for the Time Warp." *IEEE Transactions on Computers.* 41(1):68-82. January 1992.

[11] Reynolds, Jr. Paul F., Carmen M. Pancerella, and Sudhir Srinivasan. "Design and Performance Analysis of Hardware Support for Parallel Simulations." *Journal of Parallel and Distributed Computing.* 18:435-453. 1993.

[12] Comfort, John C. "The Simulation of a Master-Slave Event Set Processor," *Simulation.* 42(3):117-124. January 1989.

[13] Athanas, Peter M. and Harvey F. Silverman. "Processor Reconfiguration Through Instruction-Set Metamorphosis," *Computer.* 26(3):11-18 March 1993.

[14] Reynolds, Jr. Paul F. et. al. "Comparative Analysis of Parallel Simulation Protocols," *Proceedings of the 1989 Winter Simulation Conference.* 671-679. 1989.

[15] Perry, Douglas L. *VHDL.* New York: McGraw-Hill Inc. 1991.

[16] Kapp, Kevin L. *Partitioning Structural VHDL Circuits for Parallel Execution on Hypercubes.* MS Thesis, AFIT/GCS/ENG/93D-07, Air Force Institute of Technology (AU) Wright-Patterson AFB, OH, December 1993.

[17] Droddy, Vincent A., ed. "Chapter 5: The Mesh Connection Network and iPSC Performance Analysis," *AFIT/ENG Compendium of Parallel Programs*, Vol. I., AFIT, 1993.

[18] *Intel Paragon Supercomputer Programming II.* Intel Supercomputer Systems Division, Beaverton: Intel Corporation. August 1994

[19] *i860 Microprocessor Family Programmer's Reference Manual.* Intel Corporation Literature Sales, Mt. Prospect: 1992.

[20] *Intel Paragon Training Notes.* Intel Supercomputer Systems Division, Beaverton: Intel Corporation. March 1994.

[21] Goosby, Darin, et al, *AFIT/ENG Intel Hypercube iPSC/2 Quick Reference Manual.* Version 1.0, Air Force Institute of Technology (AU) Wright-Patterson AFB, OH , 1994.

[22] Intel Paragon OSF/1 On-line Manual Pages. Intel Supercomputer Systems Division, Beaverton: Intel Corporation. August 1994.

[23] Pickering, Ron, Clifford Addison, Jeremy Cook and David Warhurst. *Parallel Processing: A Self-Study Introduction, A First Course in Programming the iPSC/2 Hypercube*. Parallab, Dept. of Infomatics, University of Bergen, N-5020 Bergen, Norway, July 1989.

[24] *Intel Paragon User's Guide*. Intel Supercomputer Systems Division, Beaverton: Intel Corporation. June 1994.

[25] Hurford, Joel. *Accelerating Conservative Parallel Simulations of VHDL Circuits.* MS Thesis, AFIT/GCS/ENG/94D-10, Air Force Institute of Technology (AU) Wright-Patterson AFB, OH, December 1994.

Vita

Second Lieutenant Andrew C. Walton was born in Winston-Salem, North Carolina, on 13 June 1970. He graduated from Walter Williams High School in Burlington, North Carolina in 1988. He attended Marion Military Institute in Marion, Alabama in 1989 and graduated from the United States Air Force Academy in June 1993 with a Bachelor of Science Degree in Electrical Engineering. In May of 1993, Lieutenant Walton was selected to attend the Air Force Institute of Technology for completion of a Master of Science in Computer Engineering.

Permanent Address: 517 Fountain Place
Burlington, North Carolina 27215